USENIX Association

Proceedings of USENIX ATC '17

2017 USENIX Annual Technical Conference

July 12–14, 2017
Santa Clara, CA, USA

Conference Organizers

Program Co-Chairs

Dilma Da Silva, *Texas A&M University*

Bryan Ford, *École Polytechnique Fédérale de Lausanne (EPFL)*

Program Committee

Rachit Agarwal, *Cornell University*

Nadav Amit, *VMware Research Group*

Mona Attariyan, *Google*

Sorav Bansal, *Indian Institute of Technology Delhi*

Adam Bates, *University of Illinois at Urbana-Champaign*

Justin Cappos, *New York University Tandon School of Engineering*

Rong Chen, *Shanghai Jiao Tong University*

Mihai Christodorescu, *Qualcomm Research*

Charlie Curtsinger, *Grinnell College*

Christina Delimitrou, *Cornell University*

Fred Douglis, *Dell EMC*

Eric Eide, *University of Utah*

Ittay Eyal, *Cornell University*

Ada Gavrilovska, *Georgia Institute of Technology*

Vishakha Gupta, *Intel Labs*

Michio Honda, *NEC Laboratories Europe*

Yu Hua, *Huazhong University of Science and Technology*

Peng (Ryan) Huang, *Microsoft Research and Johns Hopkins University*

Taesoo Kim, *Georgia Institute of Technology*

Eddie Kohler, *Harvard University*

Jean-Pierre Lozi, *Université Nice Sophia Antipolis*

Gilles Muller, *Inria*

Donald E. Porter, *University of North Carolina at Chapel Hill*

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

Ji-Yong Shin, *Yale University*

Liuba Shrira, *Brandeis University*

Nisha Talagala, *Parallel Machines*

Chunqiang Tang, *Facebook*

Theodore Ts'o, *Google*

Dan Tsafrir, *Technion—Israel Institute of Technology*

Dan Williams, *IBM T.J. Watson Research Center*

David Wolinsky, *Facebook*

Timothy Wood, *George Washington University*

External Reviewers

Amogh Akshintala
Ghada Almashaqbeh
Brian Burg
Aleksandar Dragojevic
Sindhu Ghanta
Kartik Gopalan
Haryadi Gunawi
Istvan Haller
Sungpack Hong
Amir Hormati
Jian Huang
Michael Isard
Xin Jin
Guoliang Jin
David M. Johnson
Xiaoen Ju
Hye-Chung Kum
Patrick P. C. Lee
Cheng Li
Richard Li
Yiwen Li
Paul McKenney
Mike Mesnier
Parya Moinzadeh
Preston Moore
Michael Papamichael
Andrew Quinn
Moin Qureshi
Bharath Ramsundar
Drew Roselli
Semih Salihoglu
Igor Smolyar
LinHai Song
Christina Strong
Sing-hoi Sze
Santiago Torres-Arias
Chia-Che Tsai
Rajat Verma
Sam Weber
Gary Wong
Idan Yaniv
Yang Zhan
Tao Zhang
Yibo Zhu
Aviad Zuck

Message from the
USENIX ATC '17 Program Co-Chairs

Welcome to the 2017 USENIX Annual Technical Conference

We are awed by the effort devoted by so many in our community to making ATC '17 a success. Along the process of creating this year's program, we were exposed to high doses of technical expertise, experience in research, passion for building systems, fairness, competence, and kindness.

The incredible dedication by this year's program committee resulted in a program of 60 refereed papers and two invited talks. These papers and talks present novel research contributions and practical insights that advance the state-of-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 305 papers, of which 283 were complete submissions. The program co-chairs rejected six papers up front due to serious format violations. Of the submitted papers, 29 were short papers, which had to be at most five pages long (plus references), and the other 254 were full-length papers, which had to be at most 11 pages long plus references. We were very pleased with the high number of submissions and with the program committee's positive attitude under such heavy reviewing load.

The program committee had 35 members, including the two co-chairs. Thirteen of them had affiliations with industrial organizations, and 22 with academic organizations. The committee represented three continents and seven countries. Program committee members were allowed to submit papers. The co-chairs did not have any submissions. We followed standard rules for handling conflicts of interest: conflicted members (or co-chairs) left the room during discussion of conflicted papers. There was one paper with which both co-chairs were conflicted, and Fred Douglis (a PC member) managed its review process.

Reviewing was single-blind, done by the program committee in two rounds, with a few external reviews. In the first round, each of the 283 submitted papers received at least two reviews. Two classes of papers moved to the second round: (a) papers receiving at least one "weak accept" or better (i.e., "accept" or "strong accept") review and (b) papers where none of the reviewers rated their expertise level as "knowledgeable" or "expert." In total, 176 papers (62% of submissions) moved on. In the second round, each paper received at least two more reviews.

Altogether, we had more than 920 reviews.

After two phases of review, an online discussion was conducted among reviewers, during which the program committee decided to accept 21 highly-ranked papers, tentatively reject 85 more papers, and to further discuss 70 papers during the in-person program committee meeting.

The PC meeting was held on April 20-21 at the VMware campus in Palo Alto, CA; 31 PC members attended the meeting in person, two called in, and two could not participate. During the meeting 39 papers were accepted, resulting in the 21 papers accepted earlier in a total of 60 accepted papers. Among these 60 acceptances, three were short papers.

We added to the program two invited talks, chosen from recommendations made by members of the USENIX community. We also continued the tradition of inviting best-of-the-rest talks from the best papers at other USENIX-sponsored conferences. We have invited talks from FAST, OSDI, and USENIX Security.

We are very grateful to all who contributed to ATC'17. In addition to the authors that 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 VMware 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 '17 Program Co-Chairs
Dilma Da Silva, *Texas A&M University*
Bryan Ford, *École Polytechnique Fédérale de Lausanne (EPFL)*

USENIX ATC '17:
2017 USENIX Annual Technical Conference

Contents

Kernel

Lock-in-Pop: Securing Privileged Operating System Kernels by Keeping on the Beaten Path **1**
Yiwen Li, Brendan Dolan-Gavitt, Sam Weber, and Justin Cappos, *New York University*

Fast and Precise Retrieval of Forward and Back Porting Information for Linux Device Drivers **15**
Julia Lawall, Derek Palinski, Lukas Gnirke, and Gilles Muller, *Sorbonne Universités/UPMC/Inria/LIP6*

Optimizing the TLB Shootdown Algorithm with Page Access Tracking **27**
Nadav Amit, *VMware Research*

Falcon: Scaling IO Performance in Multi-SSD Volumes **41**
Pradeep Kumar and H. Howie Huang, *The George Washington University*

Datacenters

deTector: a Topology-aware Monitoring System for Data Center Networks **55**
Yanghua Peng, *The University of Hong Kong;* Ji Yang, *Xi'an Jiaotong University;* Chuan Wu, *The University of Hong Kong;* Chuanxiong Guo, *Microsoft Research;* Chengchen Hu, *Xi'an Jiaotong University;* Zongpeng Li, *University of Calgary*

Pricing Intra-Datacenter Networks with Over-Committed Bandwidth Guarantee **69**
Jian Guo, Fangming Liu, and Tao Wang, *Key Laboratory of Services Computing Technology and System, Ministry of Education, School of Computer Science and Technology, Huazhong University of Science and Technology;* John C.S. Lui, *The Chinese University of Hong Kong*

Unobtrusive Deferred Update Stabilization for Efficient Geo-Replication **83**
Chathuri Gunawardhana, Manuel Bravo, and Luis Rodrigues, *University of Lisbon*

Don't cry over spilled records: Memory elasticity of data-parallel applications and its application to cluster scheduling **97**
Călin Iorgulescu and Florin Dinu, *EPFL;* Aunn Raza, *NUST Pakistan;* Wajih Ul Hassan, *UIUC;* Willy Zwaenepoel, *EPFL*

Pursuing Efficiency

Popularity Prediction of Facebook Videos for Higher Quality Streaming **111**
Linpeng Tang, *Princeton University;* Qi Huang and Amit Puntambekar, *Facebook;* Ymir Vigfusson, *Emory University & Reykjavik University;* Wyatt Lloyd, *University of Southern California & Facebook;* Kai Li, *Princeton University*

Squeezing out All the Value of Loaded Data: An Out-of-core Graph Processing System with Reduced Disk I/O **125**
Zhiyuan Ai, Mingxing Zhang, and Yongwei Wu, *Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology* (*TNLIST*)*, Tsinghua University and Research Institute of Tsinghua;* Xuehai Qian, *University of Southern California;* Kang Chen and Weimin Zheng, *Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology* (*TNLIST*)*, Tsinghua University, and Research Institute of Tsinghua*

Ending the Anomaly: Achieving Low Latency and Airtime Fairness in WiFi **139**
Toke Høiland-Jørgensen, *Karlstad University;* Michał Kazior, *Tieto Poland;* Dave Täht, *TekLibre;* Per Hurtig and Anna Brunstrom, *Karlstad University*

Persona: A High-Performance Bioinformatics Framework **153**
Stuart Byma and Sam Whitlock, *EPFL;* Laura Flueratoru, *University Politehnica of Bucharest;* Ethan Tseng, *CMU;* Christos Kozyrakis, *Stanford University;* Edouard Bugnion and James Larus, *EPFL*

Let's Talk about GPUs

SPIN: Seamless Operating System Integration of Peer-to-Peer DMA Between SSDs and GPUs **167**
Shai Bergman and Tanya Brokhman, *Technion;* Tzachi Cohen, *unaffiliated;* Mark Silberstein, *Technion*

Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters **181**
Hao Zhang, *Carnegie Mellon University;* Zeyu Zheng, *Petuum Inc.;* Shizhen Xu and Wei Dai, *Carnegie Mellon University;* Qirong Ho, *Petuum Inc.;* Xiaodan Liang, Zhiting Hu, Jinliang Wei, and Pengtao Xie, *Carnegie Mellon University;* Eric P. Xing, *Petuum Inc.*

Garaph: Efficient GPU-accelerated Graph Processing on a Single Machine with Balanced Replication **195**
Lingxiao Ma, Zhi Yang, and Han Chen, *Computer Science Department, Peking University, Beijing, China;* Jilong Xue, *Microsoft Research, Beijing, China;* Yafei Dai, *Institute of Big Data Technologies Shenzhen Key Lab for Cloud Computing Technology & Applications, School of Electronics and Computer Engineering* (*SECE*), *Peking University, Shenzhen, China*

GPU Taint Tracking **209**
Ari B. Hayes, *Rutgers University;* Lingda Li, *Brookhaven National Laboratory;* Mohammad Hedayati, *University of Rochester;* Jiahuan He and Eddy Z. Zhang, *Rutgers University;* Kai Shen, *Google*

Virtualization

Optimizing the Design and Implementation of the Linux ARM Hypervisor **221**
Christoffer Dall, Shih-Wei Li, and Jason Nieh, *Columbia University*

Multi-Hypervisor Virtual Machines: Enabling an Ecosystem of Hypervisor-level Services **235**
Kartik Gopalan, Rohit Kugve, Hardik Bagdi, and Yaohui Hu, *Binghamton University;* Daniel Williams and Nilton Bila, *IBM T.J. Watson Research Center*

Preemptive, Low Latency Datacenter Scheduling via Lightweight Virtualization **251**
Wei Chen, *University of Colorado, Colorado Springs;* Jia Rao, *University of Texas at Arlington;* Xiaobo Zhou, *University of Colorado, Colorado Springs*

The RCU-Reader Preemption Problem in VMs **265**
Aravinda Prasad and K Gopinath, *Indian Institute of Science, Bangalore;* Paul E. McKenney, *IBM Linux Technology Center, Beaverton*

Security and Privacy I

Bunshin: Compositing Security Mechanisms through Diversification **271**
Meng Xu, Kangjie Lu, Taesoo Kim, and Wenke Lee, *Georgia Institute of Technology*

Glamdring: Automatic Application Partitioning for Intel SGX **285**
Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, and Florian Kelbert, *Imperial College London;* Tobias Reiher, *TU Dresden;* David Goltzsche, *TU Braunschweig;* David Eyers, *University of Otago;* Rudiger Kapitza, *TU Braunschweig;* Christof Fetzer, *TU Dresden;* Peter Pietzuch, *Imperial College London*

High-Resolution Side Channels for Untrusted Operating Systems **299**
Marcus Hähnel, *TU Dresden, Operating Systems Group;* Weidong Cui and Marcus Peinado, *Microsoft Research*

Understanding Security Implications of Using Containers in the Cloud **313**
Byungchul Tak, *Kyungpook National University;* Canturk Isci, Sastry Duri, Nilton Bila, Shripad Nadgowda, and James Doran, *IBM TJ Watson Research Center*

(continued on next page)

Key-Value Stores and Databases

Memshare: a Dynamic Multi-tenant Key-value Cache **321**
Asaf Cidon, *Stanford University;* Daniel Rushton, *University of Utah;* Stephen M. Rumble, *Google Inc.;* Ryan Stutsman, *University of Utah*

Replication-driven Live Reconfiguration for Fast Distributed Transaction Processing **335**
Xingda Wei, Sijie Shen, Rong Chen, and Haibo Chen, *Shanghai Jiao Tong University*

HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems **349**
Fei Xia, *Institute of Computing Technology, Chinese Academy of Sciences; University of Chinese Academy of Sciences;* Dejun Jiang, Jin Xiong, and Ninghui Sun, *Institute of Computing Technology, Chinese Academy of Sciences*

TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores **363**
Oana Balmau, Diego Didona, Rachid Guerraoui, and Willy Zwaenepoel, *EPFL;* Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka, *Nutanix*

Help Me Debug

Engineering Record And Replay For Deployability **377**
Robert O'Callahan and Chris Jones, *unaffiliated;* Nathan Froyd, *Mozilla Corporation;* Kyle Huey, *unaffiliated;* Albert Noll, *Swisscom AG;* Nimrod Partush, *Technion*

Proactive error prediction to improve storage system reliability **391**
Farzaneh Mahdisoltani, *University of Toronto;* Ioan Stefanovici, *Microsoft Research;* Bianca Schroeder, *University of Toronto*

Towards Production-Run Heisenbugs Reproduction on Commercial Hardware **403**
Shiyou Huang, Bowen Cai, and Jeff Huang, *Texas A&M University*

A DSL Approach to Reconcile Equivalent Divergent Program Executions **417**
Luís Pina, Daniel Grumberg, Anastasios Andronidis, and Cristian Cadar, *Imperial College London*

Networking

Titan: Fair Packet Scheduling for Commodity Multiqueue NICs **431**
Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael Swift, *UW-Madison*

MopEye: Opportunistic Monitoring of Per-app Mobile Network Performance **445**
Daoyuan Wu, *Singapore Management University;* Rocky K. C. Chang, Weichao Li, and Eric K. T. Cheng, *The Hong Kong Polytechnic University;* Debin Gao, *Singapore Management University*

Emu: Rapid Prototyping of Networking Services **459**
Nik Sultana, Salvator Galea, David Greaves, Marcin Wojcik, and Jonny Shipton, *University of Cambridge;* Richard Clegg, *Queen Mary University of London;* Luo Mai, *Imperial College London;* Pietro Bressana and Robert Soule, *Università della Svizzera italiana;* Richard Mortier, *University of Cambridge;* Paolo Costa, *Microsoft Research;* Peter Pietzuch, *Imperial College London;* Jon Crowcroft, Andrew W Moore, and Noa Zilberman, *University of Cambridge*

Protego: Cloud-Scale Multitenant IPsec Gateway **473**
Jeongseok Son, *KAIST, Microsoft Research;* Yongqiang Xiong, *Microsoft Research;* Kun Tan, *Huawei;* Paul Wang and Ze Gan, *Microsoft Research;* Sue Moon, *KAIST*

Caching along the Way

Cache Modeling and Optimization using Miniature Simulations **487**
Carl Waldspurger, Trausti Saemundson, and Irfan Ahmad, *CachePhysics, Inc.;* Nohhyun Park, *Datos IO, Inc.*

Hyperbolic Caching: Flexible Caching for Web Applications **499**
Aaron Blankstein, *Princeton University;* Siddhartha Sen, *Microsoft Research;* Michael J. Freedman, *Princeton University*

Execution Templates: Caching Control Plane Decisions for Strong Scaling of Data Analytics **513**
Omid Mashayekhi, Hang Qu, Chinmayee Shah, and Philip Levis, *Stanford University*

cHash: Detection of Redundant Compilations via AST Hashing **527**
Christian Dietrich and Valentin Rothberg, *Leibniz Universität Hannover;* Ludwig Füracker and Andreas Ziegler, *Friedrich-Alexander Universität Erlangen-Nürnberg;* Daniel Lohmann, *Leibniz Universität Hannover*

Storage

Giza: Erasure Coding Objects across Global Data Centers **539**
Yu Lin Chen, *NYU & Microsoft Corporation;* Shuai Mu and Jinyang Li, *NYU;* Cheng Huang, Jin Li, Aaron Ogus, and Douglas Phillips, *Microsoft Corporation*

SmartCuckoo: A Fast and Cost-Efficient Hashing Index Scheme for Cloud Storage Systems **553**
Yuanyuan Sun and Yu Hua, *Huazhong University of Science and Technology;* Song Jiang, *University of Texas, Arlington;* Qiuyu Li, Shunde Cao, and Pengfei Zuo, *Huazhong University of Science and Technology*

Repair Pipelining for Erasure-Coded Storage **567**
Runhui Li, Xiaolu Li, Patrick P. C. Lee, and Qun Huang, *The Chinese University of Hong Kong*

PARIX: Speculative Partial Writes in Erasure-Coded Systems **581**
Huiba Li, *mos.meituan.com;* Yiming Zhang, *NUDT;* Zhiming Zhang, *mos.meituan.com;* Shengyun Liu, Dongsheng Li, Xiaohui Liu, and Yuxing Peng, *NUDT*

Multicore

E-Team: Practical Energy Accounting for Multi-Core Systems **589**
Till Smejkal and Marcus Hähnel, *TU Dresden;* Thomas Ilsche, *Center for Information Services and High Performance Computing (ZIH) Technische Universität Dresden;* Michael Roitzsch, *TU Dresden;* Wolfgang E. Nagel, *Center for Information Services and High Performance Computing (ZIH) Technische Universität Dresden;* Hermann Härtig, *TU Dresden*

Scalable NUMA-aware Blocking Synchronization Primitives **603**
Sanidhya Kashyap, Changwoo Min, and Taesoo Kim, *Georgia Institute of Technology*

StreamBox: Modern Stream Processing on a Multicore Machine **617**
Hongyu Miao and Heejin Park, *Purdue ECE;* Myeongjae Jeon and Gennady Pekhimenko, *Microsoft Research;* Kathryn S. McKinley, *Google;* Felix Xiaozhu Lin, *Purdue ECE*

Everything you always wanted to know about multicore graph processing but were afraid to ask **631**
Jasmina Malicevic, Baptiste Lepers, and Willy Zwaenepoel, *EPFL*

Security and Privacy II

Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX **645**
Chia-Che Tsai, *Stony Brook University;* Donald E. Porter, *University of North Carolina at Chapel Hill and Fortanix;* Mona Vij, *Intel Corporation*

PrivApprox: Privacy-Preserving Stream Analytics **659**
Do Le Quoc and Martin Beck, *TU Dresden;* Pramod Bhatotia, *University of Edinburgh;* Ruichuan Chen, *Nokia Bell Labs;* Christof Fetzer and Thorsten Strufe, *TU Dresden*

Mercury: Bandwidth-Effective Prevention of Rollback Attacks Against Community Repositories **673**
Trishank Karthik Kuppusamy, Vladimir Diaz, and Justin Cappos, *New York University*

(continued on next page)

CAB-Fuzz: Practical Concolic Testing Techniques for COTS Operating Systems **689**
Su Yong Kim, *The Affiliated Institute of ETRI;* Sangho Lee, Insu Yun, and Wen Xu, *Georgia Tech;* Byoungyoung Lee, *Purdue University;* Youngtae Yun, *The Affiliated Institute of ETRI;* Taesoo Kim, *Georgia Tech*

Don't Forget the Memory

Log-Structured Non-Volatile Main Memory **703**
Qingda Hu, *Tsinghua University;* Jinglei Ren and Anirudh Badam, *Microsoft Research;* Jiwu Shu, *Tsinghua University;* Thomas Moscibroda, *Microsoft Research*

Soft Updates Made Simple and Fast on Non-volatile Memory **719**
Mingkai Dong and Haibo Chen, *Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University*

SmartMD: A High Performance Deduplication Engine with Mixed Pages 733
Fan Guo, *University of Science and Technology of China;* Yongkun Li, *University of Science and Technology of China; Collaborative Innovation Center of High Performance Computing, NUDT;* Yinlong Xu, *University of Science and Technology of China; Anhui Province Key Laboratory of High Performance Computing, USTC;* Song Jiang, *University of Texas, Arlington;* John C. S. Lui, *The Chinese University of Hong Kong*

Elastic Memory Management for Cloud Data Analytics **745**
Jingjing Wang and Magdalena Balazinska, *University of Washington*

File Systems

Improving File System Performance of Mobile Storage Systems Using a Decoupled Defragmenter **759**
Sangwook Shane Hahn, *Seoul National University;* Sungjin Lee, *Daegu Gyeongbuk Institute of Science and Technology;* Cheng Ji, *City University of Hong Kong;* Li-Pin Chang, *National Chiao-Tung University;* Inhyuk Yee, *Seoul National University;* Liang Shi, *Chongqing University;* Chun Jason Xue, *City University of Hong Kong;* Jihong Kim, *Seoul National University*

Octopus: an RDMA-enabled Distributed Persistent Memory File System **773**
Youyou Lu, Jiwu Shu, and Youmin Chen, *Tsinghua University;* Tao Li, *University of Florida*

iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call **787**
Daejun Park and Dongkun Shin, *Sungkyunkwan University, Korea*

Scaling Distributed File Systems in Resource-Harvesting Datacenters **799**
Pulkit A. Misra, *Duke University;* Íñigo Goiri, Jason Kace, and Ricardo Bianchini, *Microsoft Research*

Lock-in-Pop: Securing Privileged Operating System Kernels by Keeping on the Beaten Path

Yiwen Li Brendan Dolan-Gavitt Sam Weber Justin Cappos
New York University

Abstract

Virtual machines (VMs) that try to isolate untrusted code are widely used in practice. However, it is often possible to trigger zero-day flaws in the host Operating System (OS) from inside of such virtualized systems. In this paper, we propose a new security metric showing strong correlation between "popular paths" and kernel vulnerabilities. We verify that the OS kernel paths accessed by popular applications in everyday use contain significantly fewer security bugs than less-used paths. We then demonstrate that this observation is useful in practice by building a prototype system which ***locks*** an application into using only ***popular*** OS kernel paths. By doing so, we demonstrate that we can prevent the triggering of zero-day kernel bugs significantly better than three other competing approaches, and argue that this is a practical approach to secure system design.

1 Introduction

The number of attacks involving the exploitation of zero-day vulnerabilities more than doubled from 2014 to 2015 [52]. Skilled hackers can find a security flaw in a system and use it to hold the system's users hostage, e.g., by gaining root access and compromising the host [25]. Similarly, zero-day vulnerabilities can be exploited [17] or their presence not be acknowledged [30] by government agencies, thus rendering millions of devices vulnerable.

In theory, running a program in an operating-system-level virtual machine (OSVM) like Docker [15] or LXC [28] should prevent bugs in the host OS kernel from triggering. However, the isolation provided by such systems is not the whole answer and faces some significant drawbacks. To be effective, the OSVM's software must not contain any bugs that could allow the program to escape the machine's containment and interact directly with the host OS. Unfortunately, these issues are very common in OSVMs, with 14 CVE vulnerabilities confirmed for Docker [14] since 2014. The large amount of complex code needed to run such a system increases the odds that flaws will be present, and, in turn, that tens of millions of user machines could be at risk [25]. Furthermore, isolation will not work if a malicious program can access even a small portion of the host OS's kernel that contains a zero-day flaw [12]. Both of these drawbacks reveal the key underlying weakness in designing OSVM systems – a lack of information as to which parts of the host kernel can be safely exported to user programs.

Several attempts have been made to find a reliable metric to pinpoint where bugs are most likely to be in kernel code. A number of previous studies have suggested that older code may be less vulnerable than new code [32] or that certain parts (such as device drivers) of the kernel [10] may be more bug-prone than others. To these hypotheses, we add a new security metric idea, called "popular paths." Positing that bugs in the popular paths, associated with frequently-used programs, are more likely to be found in software testing because of the numerous times they are executed by diverse pieces of software, we propose that kernel code found in these paths would have less chance of containing bugs than code in less-used parts of the kernel. We perform a quantitative analysis of resilience to flaws in two versions of the Linux kernel (version 3.13.0 and version 3.14.1), and find that only about 3% of the bugs are present in popular code paths, despite these paths accounting for about one-third of the total reachable kernel code. When we test our "popular paths" metric against the two aforementioned "code age" and "device drivers" metrics, we find our "popular paths" metric is much more effective (Section 3.2).

This key information inspired the idea that if we could design virtual machines that use only "popular kernel paths," a strategy we have dubbed *Lock-in-Pop*, it would greatly increase resilience to zero-day bugs in the host OS kernel. Yet using such a design scheme creates a few challenges that would need to be overcome. These include:

- It might not be possible in real-life codebases to completely avoid "unpopular paths." If other applications, or future versions of applications we tested, frequently require the use of "unpopular paths," would this make our metric untenable? (Section 4.2)

- The exploits that adversaries use change over time. Could our observation that "popular paths" are safer be only an artifact of when we did our measurements, and not be predictive of future exploits? (Section 3.2)

- Lastly, can developers make use of this observation in a practical setting? That is, is it feasible for developers to actively try to avoid unpopular code paths? (Section 4.3)

While we address some of these challenges in developing the *Lock-in-Pop* design, we want to test how well a system could function if it forced applications to use only popular kernel paths. To conduct these tests, we built a prototype system, called Lind. For Lind, we pick two key components – Google's Native Client (NaCl) [51] and Seattle's Repy [8]. NaCl serves as a computational module that isolates binaries, providing memory safety for legacy programs running in our OSVM. It also passes system calls invoked by the program to the operating system interface, called SafePOSIX. SafePOSIX re-creates the broader POSIX functionalities needed by applications, while being contained within the Repy sandbox. An API in the sandbox only allows access to popular kernel paths, while the small (8K LOC) sandbox kernel of Repy isolates flaws in SafePOSIX to prevent them from directly accessing the host OS kernel.

To test the effectiveness of Lind and our "popular paths" metric, we replicated 35 kernel bugs discovered in Linux kernel version 3.14.1. We attempted to trigger those bugs in Lind and three other virtualized environments, including Docker [15], LXC [28], and Graphene [43]. In this study, our evaluation was focused on comparing operating-system-level virtualization containers, such as Docker and LXC, and library OSes, such as Graphene. We excluded bare-metal hypervisors [4, 46], hardware-based virtualization [3, 22] and full virtualization virtual machines, such as VirtualBox [45], VMWare Workstation [47], and QEMU [37]. While our "popular paths" metric may potentially apply to those systems, a direct comparison is not possible since they have different ways of accessing hardware resources, and would require different measurement approaches.

Our results show that applications in Lind are substantially less likely to trigger kernel bugs. By doing so, we demonstrate that forcing an application to use only popular OS paths can be an effective and practical method to improve system security. Armed with this knowledge, the *Lock-in-Pop* principle can be adapted to incorporate other OSVM design configurations.

In summary, the main contributions of this paper are as follows:

- We propose a quantitative metric that evaluates security at the line-of-code level, and verify our hypothesis that "popular paths" have significantly fewer security bugs than other paths.
- Based on the "popular paths" metric, we develop a new design scheme called *Lock-in-Pop* that accesses only popular code paths through a very small trusted computing base. The need for complex functionality is addressed by re-creating riskier system calls in a memory-safe programming language within a secure sandbox.
- To demonstrate the practicality of the "popular paths" metric, we build a prototype virtual machine, Lind, using the *Lock-in-Pop* design, and test its effectiveness against three other virtual machines. We find that Lind exposes 8-12x fewer zero-day kernel bugs.

2 Goals and Threat Model

In this section, we define the scope of our efforts. We also briefly note why this study does not evaluate a few existing design schemes.

Goals. Ultimately, our goal is to help designers create systems that allow untrusted programs to run on unpatched and vulnerable host OSes without triggering vulnerabilities that attackers could exploit. Developing effective defenses for the host OS kernel is essential as kernel code can expose privileged access to attackers that could lead to a system takeover.

Our hypothesis is that OS kernel code paths that are frequently used receive more attention and therefore are less likely to contain security vulnerabilities. Our approach will be to test this hypothesis and explore the feasibility of building more secure virtualization systems, such as guest OSVMs, system call interposition modules, and library OSes, by forcing untrusted applications to stay on popular kernel code paths.

Threat model. When an attack attempt is staged on a host OS in a virtualization system, the exploit can be done either directly or indirectly. In a direct exploit, the attacker accesses a vulnerable portion of the host OS's kernel using crafted attack code. In an indirect exploit, the attacker first takes advantage of a vulnerability in the virtualization system itself (for example, a buffer overflow vulnerability) to escape the VM's containment. Once past the containment, the attacker can run arbitrary code in the host OS. The secure virtualization system design we propose in Section 4 can prevent both types of attacks effectively.

Based on the goals mentioned above, we make the following assumptions about the potential threats our system could face:

- The attacker possesses knowledge of one or more unpatched vulnerabilities in the host OS.
- The attacker can execute any code in the secure virtualization system.
- If the attack program can trigger a vulnerability in any privileged code, whether in the host OS or the secure virtualization system, the attacker is then considered successful in compromising the system.

3 Developing a Quantitative Metric for Evaluating Kernel Security

If we knew which lines of code in the kernel are likely to contain zero-day bugs, we could try to avoid using them in an OSVM. In this section, we formulate and test a quantitative evaluation metric that can indicate which lines of code are likely to contain bugs. This metric is based on the idea that kernel paths executed by popular applications during everyday use are less likely to contain security flaws. The rationale is that these code paths are well-tested due to their constant use, and thus fewer bugs can go undetected. Our initial tests yielded promising results. Additionally, when tested against two earlier strategies for predicting bug locations in the OS kernel, our metric compared favorably.

3.1 Experimental Setup

We used two different versions of the Linux kernel in our study. Since our findings for these versions are quantitatively and qualitatively similar, we report the results for 3.13.0 in this section and use 3.14.1 in Section 5. To trace the kernel, we used `gcov` [19], a standard program profiling tool in the GCC suite. The tool indicates which lines of kernel code are executed when an application runs.

Popular kernel paths. To capture the popular kernel paths, we used two strategies concurrently. First, we attempted to capture the normal usage behavior of popular applications. To do this, two students used applications from the 50 most popular packages in Debian 7.0 (omitting libraries, which are automatically included by packages that depend on them) according to the Debian Popularity Contest [1], which tracks the usage of Debian packages on an opt-in basis. Each student used 25 applications for their tasks (e.g., writing, spell checking, printing in a text editor, or using an image processing program). These tests were completed over 20 hours of total use over 5 calendar days.

The second strategy was to capture the total range of applications an individual computer user might regularly access. The students used the workstation as their desktop machine for a one-week period. They did their homework, developed software, communicated with friends and family, and so on, using this system. Software was installed as needed. From these two strategies, we obtained a profile of the lines of kernel code that defined our popular kernel paths. We make these traces publicly available to other researchers [24], so they may analyze or replicate our results.

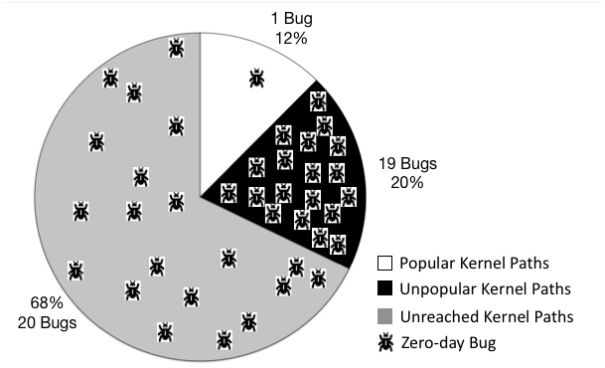


Figure 1: Percentage of different kernel areas that were reached during LTP and Trinity system call fuzzing experiments, with the zero-day kernel bugs identified in each area.

Reachable kernel paths. There are certain paths in the kernel, such as unloaded drivers, that are unreachable and unused. To determine which paths are unreachable, we used two techniques. First, we performed system call fuzzing with the Trinity system call fuzz tester [42]. Second, we used the Linux Test Project (LTP) [26], a test suite written with detailed kernel knowledge.

Locating bugs. Having identified the kernel paths used in popular applications, we then investigated how bugs are distributed among these paths. We collected a list of severe kernel bugs from the National Vulnerability Database [31]. For each bug, we found the patch that fixed the problem and identified which lines of kernel code were modified to remove it. For the purpose of this study, a user program that can execute a line of kernel code changed by such a patch is considered to have the *potential to exploit that flaw*. Note that it is possible that, in some situations, this will over-estimate the exploitation potential because reaching the lines of kernel code where a bug exists does not necessarily imply a reliable, repeatable capability to exploit the bug.

3.2 Results and Analysis

Bug distribution. The experimental results from Section 3.1 show that only one of the 40 kernel bugs tested for was found among the popular paths, even though these paths make up 12.4% of the kernel (Figure 1).

To test the significance of these results, we performed a power analysis. We assume that kernel bugs appear at

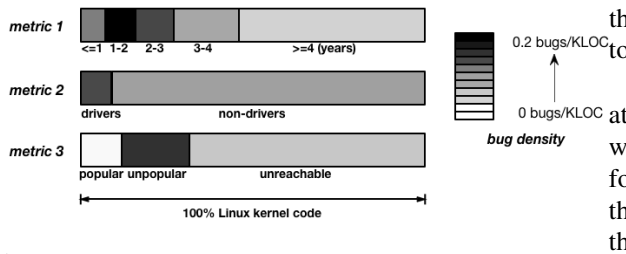


Figure 2: Bug density comparison among three metrics.

an average rate proportional to the number of lines of kernel code. Therefore, consistent with prior research [29], the rate of defect occurrence per LOC follows a Poisson distribution [35]. The premise we tested is that bugs occur at different rates in different parts of the kernel, i.e., that the less popular kernel portion has more bugs.

We first divided the kernel into two sets, A and B, where bugs occur at rates λ_A and λ_B, and $\lambda_A \neq \lambda_B$. In this test, A represents the popular paths in the kernel, while B addresses the less commonly-used paths. Given the null hypothesis that the rate of defect occurrences is the *same* in set A and B (or bugs in A and B are drawn from the same Poisson distribution), we used the Uniformly Most Powerful Unbiased (UMPU) test [39] to compare unequal-sized code blocks. At a significance level of $\alpha = 0.01$, the test was significant at $p = 0.0015$, rejecting the null hypothesis. The test also reported a 95% confidence that $\lambda_A/\lambda_B \in [0.002, 0.525]$. This indicates that the ratio between the bug rates is well below 1. Since B has a bug rate much larger than that of A, this result shows that popular paths have a much lower bug rate than unpopular ones.

Comparison with other security metrics. Ozment, et al. [32] demonstrated that older code in the Berkeley Software Distribution (BSD) [7] kernel tended to have fewer bugs (metric 1). To test Ozment's metric using our Linux bug dataset, we separated the code into five different age groups. Our results (Figure 2) showed a substantial number of bugs located in each group, and not just in the newer code. Therefore, buggy code in the Linux kernel cannot be identified simply by this age-based metric. In addition, this metric would seem to have limited use for designing a secure virtualization system, as no system could run very long exclusively on old code.

Another metric, reported by Chou, et al. [10], showed that certain parts of the kernel, particularly device drivers, were more vulnerable than others (metric 2). Applying this metric on our dataset, we found that the driver code in our version of the Linux kernel accounted for only 8.9% of the total codebase, and contained just 4 out of the 40 bugs (Figure 2). One reason for this is that after Chou's study was published system designers focused efforts on improving driver code. Palix [33] found that `drivers` now has a lower fault rate than other directories, such as `arch` and `fs`.

Additionally, there are other security metrics that operate at a coarser granularity, e.g., the file level. However, when our kernel tests were run at a file granularity, we found that even popular programs used parts of 32 files that contained flaws. Yet, only one bug was triggered by those programs. In addition, common programs tested at this level also executed 36 functions that were later patched to fix security flaws, indicating the need to localize bugs at a finer granularity.

To summarize, our results demonstrate that previously proposed security metrics show only weak correlation between the occurrence of bugs and the type of code they target. In contrast, our metric (metric 3) provides an effective and statistically significant means for predicting where in the kernel exploitable flaws will likely be found. For the remainder of the paper, we will focus on using our "popular paths" metric to design and build secure virtualization systems.

4 A New Design for Secure Virtualization Systems

In the previous section we have shown that "popular paths" correlate in a statistically significant manner with security. Next, we want to demonstrate that our "popular paths" metric is useful in practice for designing secure virtualization systems. We first briefly discuss the limitations faced by existing methods, due to the lack of a good security metric. We then discuss our new design scheme named *Lock-in-Pop*, which follows our metric by accessing only popular code paths.

4.1 Previous Attempts and Their Limitations

System call interposition (SCI). SCI systems [20, 48] filter system calls to mediate requests from untrusted user code instead of allowing them to go directly to the kernel. The filter checks a predefined security policy to decide which system calls are allowed to pass to the underlying kernel, and which ones must be stopped.

This design is limited by its overly complicated approach to policy decisions and implementation. To make a policy decision, the system needs to obtain and interpret the OS state (e.g., permissions, user groups, register flags) associated with the programs it is monitoring. The complexity of OS states makes this process difficult and can lead to inaccurate policy decisions.

Functionality re-creation. Systems such as Drawbridge [36], Bascule [5], and Graphene [43] can provide richer functionality and run more complex programs than most systems built with SCI alone because they have their own interfaces and libraries. We label such a design as "functionality re-creation."

The key to this design is to not fully rely on the underlying kernel for system functions, but to re-create its own

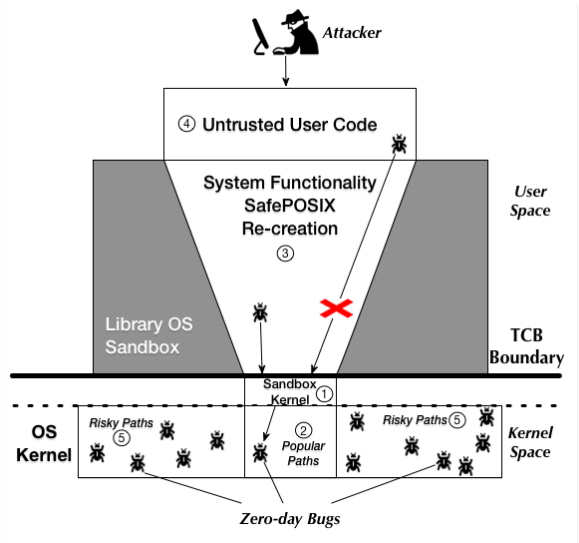


Figure 3: *Lock-in-Pop* design ensures safe execution of untrusted user code despite existing potential zero-day bugs in the OS kernel.

system functionality. When it has to access resources, like memory, CPU, and disk storage, the system accesses the kernel directly with its underlying TCB code.

Functionality re-creation provides a more realistic solution to building virtualization systems than earlier efforts. However, functionality re-creation has two pitfalls: first, if the re-created functionality resides in the TCB of the virtualization system, then vulnerabilities there can expose the host OS to attack as well. For example, hundreds of vulnerabilities have been reported in existing virtualization systems, such as QEMU and VMWare, over the past ten years [31].

Second, functionality re-creation may assume that the underlying host kernel is correct. As we have seen, this assumption is often incorrect: host kernels may have bugs in their implementation that leave them vulnerable to attack. Thus, to provide the greatest assurance that the host kernel will not be exposed to malicious user programs, a secure functionality re-creation design should try to deliberately avoid kernel paths that are likely to contain flaws. We discuss this approach in detail next.

4.2 Lock-in-Pop: Staying on the Beaten Path

Recall that we want to show that the "popular paths" metric can be used in practice. We do so by devising a design in which all code, including the complex part of the operating system interface, accesses only popular kernel paths through a small TCB. As it "locks" all functionality requests into only the "popular paths," we call this design *Lock-in-Pop*.

At the lowest level of the design (interfacing with the host OS) is the sandbox kernel (① in Figure 3). The sandbox kernel's main role is to ensure that only popular paths (② in Figure 3) of the host OS's kernel can be accessed. The sandbox kernel could thus function as a very granular system call filter, or as the core of a programming language sandbox. Note that the functionality provided by the sandbox kernel is (intentionally) much less than what an application needs. For example, an application may store files in directories and set permissions on those files. The sandbox kernel may provide a much simpler abstraction (e.g., a block storage abstraction), so long as the strictly needed functionality (e.g., persistent storage) is provided.

Constructing the sandbox kernel is not dependent on any specific technique or programming language. Instead, the sandbox kernel follows a central design principle to include only simple and necessary system calls with basic flags, which can be checked to verify that only "popular paths" are used. The sandbox kernel should start with building-block functions to first form a minimum set of system calls. To give one example, for network programs, opening a TCP connection would be considered an essential function. We can verify that the lines of kernel code that correspond to opening TCP sockets, such as lines in `void tcp_init_sock(struct sock *sk)`, are included in the "popular paths" for that system, and so decide to include the `open_tcp_connection()` function in the sandbox kernel. Examples of other necessary functions are `file.open`, `file.close`, `file.read`, and `file.write` for filesystem functions, and `create_thread`, `create_lock`, `lock.acquire`, and `lock.release` for threading functions.

In order to make security our priority, the designed sandbox kernel should only use a subset of the "popular paths." For systems where security is not as critical, trade-offs can certainly be made to include some "unpopular paths" to accommodate applications. Further discussion of this trade-off is beyond the scope of this paper, though we acknowledge it is an issue that should be addressed as *Lock-in-Pop* is deployed. While restricting the system call interface is a big hammer for limiting access to "popular paths" in the kernel, we believe that this is the best choice available, given that we do not want to require modification to the kernel, and would like to allow users to easily run their applications without much extra effort.

The application is provided more complex functionality via the SafePOSIX re-creation (③ in Figure 3). SafePOSIX has the needed complexity to build more convenient higher-level abstractions using the basic functionality provided by the sandbox kernel. The SafePOSIX re-creation is itself isolated within a library OS sandbox, which forces all system calls to go through the sandbox kernel. So long as this is performed, all calls from the SafePOSIX re-creation will only touch the permitted

(popular) kernel paths in the underlying host OS.

Similarly, untrusted user code (④ in Figure 3) also must be restricted in the way in which it performs system calls. System calls must go through the SafePOSIX re-creation, into the sandbox kernel, and then to the host OS. This is done because if user code could directly make system calls, it could access any desired path in the host OS's kernel, and thus exploit bugs within it.

Note that it is expected that bugs will occur in many components, including both the non-popular (risky) kernel paths (⑤ in Figure 3), and in the SafePOSIX re-creation. Even the user program will be buggy or perhaps explicitly malicious (created by attackers). Since the remaining components (① and ② in Figure 3) are small and can be thoroughly tested, this leads to a lower risk of compromise.

4.3 Implementation of Lock-in-Pop

To test the practicality of the "popular paths" metric and our *Lock-in-Pop* design, we implement a prototype virtual machine called Lind.[1] The purpose of building the Lind prototype is to demonstrate that our "popular paths" metric is practical, and that developers can build secure systems using it. Lind is divided into a *computational module* that enforces software fault isolation (SFI) and a *SafePOSIX module* that safely re-creates the OS functionality needed by user applications. We use a slightly modified version of Native Client (NaCl) [51] for the computational module; SafePOSIX is implemented using Restricted Python (Repy) [8] and supports complex user applications without exposing potentially risky kernel paths.

In this section we provide a brief description of these components and how they were integrated into Lind, followed by an example of how the system works.

4.3.1 Primary Components

Native Client. We use NaCl to isolate the computation of the user application from the kernel. NaCl allows Lind to work on most types of legacy code. It compiles the programs to produce a binary with software fault isolation. This prevents applications from performing system calls or executing arbitrary instructions. Instead, the application will call into a small, privileged part of NaCl that forwards system calls. In NaCl's original implementation, these calls would usually be forwarded to the host OS kernel. In Lind, we modified NaCl to instead forward these calls to our SafePOSIX re-creation (described in detail below).

Repy Sandbox. To build an API that can access the safe parts of the underlying kernel while still supporting existing applications, we need two things. First, we need a restricted sandbox kernel that only allows access to popular kernel paths. We used Seattle's Repy [8] sandbox to perform this task. Second, we have to provide complex system functions to user programs. For this task we created SafePOSIX, which implements the widely accepted standard POSIX interface on top of Repy.

Because the sandbox kernel is the only code that will be in direct contact with host system calls, it should be small (to make it easy to audit), while providing primitives that can be used to build more complex functionality. We used Seattle's Repy system API due to its tiny (around 8K LOC) sandbox kernel and its minimal set of system call APIs needed to build general computational functionality. Repy allows access only to the popular portions of the OS kernel through 33 basic API functions, including 13 network functions, 6 file functions, 6 threading functions, and 8 miscellaneous functions (Table 1) [8, 38].

Repy is only one possible implementation of the sandbox kernel built for our *Lock-in-Pop* design. It was chosen because it starts with basic building-block functions and tries to be conservative in what underlying kernel functionality it uses. Repy was designed and implemented before our "popular paths" study, and so it was not a perfect match, but it we experimentally verified that it uses a subset of the "popular paths." As reported in our evaluation (Section 5.3), Repy accessed a subset (around 70% to 80%) of the "popular paths."

Our current implementation does not end up using all of the "popular paths." It is certainly safe to use fewer paths than are available, but it is possible that we are missing out on some performance or compatibility gains. As we extend our prototype, the "popular path" metric will allow us to check whether new APIs we add expose potentially unsafe kernel code to applications in the sandbox.

4.3.2 Enhanced Safety in Call Handling with SafePOSIX Re-creation

The full kernel interface is extremely rich and hard to protect. The *Lock-in-Pop* design used to build Lind provides enhances safety protection through both isolation and a POSIX interface (SafePOSIX). The latter re-creates risky system calls to provide full-featured API for legacy applications, with minimal impact on the kernel.

In Lind, a system call issued from user code is received by NaCl, and then redirected to SafePOSIX. To service a system call in NaCl, a server routine in Lind marshals its arguments into a text string, and sends the call and the arguments to SafePOSIX. The SafePOSIX re-creation services the system call request, marshals the result, and returns it back to NaCl. Eventually, the result is returned as the appropriate native type to the calling program.

SafePOSIX is safe because of two design principles. First, its re-creation only relies on a small set of basic Repy functions (Table 1). Therefore, the interac-

[1] Lind is an old English word for a lightweight, but still strong shield constructed from two layers of linden wood.

Repy Function	Available System Calls
Networking	*gethostbyname, openconnection, getmyip, socket.send, socket.receive, socket.close, listenforconnection, tcpserversocket.getconnection, tcpserversocket.close, sendmessage, listenformessage, udpserversocket.getmessage, and udpserversocket.close.*
File System I/O Operations	*openfile(filename, create), file.close(), file.readat(size limit, offset), file.writeat(data, offset), listfiles(), and removefile(filename).*
Threading	*createlock, sleep, lock.acquire, lock.release, createthread, and getthreadname.*
Miscellaneous Functions	*getruntime, randombytes, log, exitall, createvirtualnamespace, virtualnamespace.evaluate, getresources, and getlasterror.*

Table 1: Repy sandbox kernel functions that support Lind's SafePOSIX re-creation.

tion with the host OS kernel is strictly controlled. Second, the SafePOSIX re-creation is run within the Repy programming language sandbox, which properly isolates any bugs inside SafePOSIX itself.

5 Evaluation

To demonstrate that our "popular paths" metric is useful and practical, we used our Lind prototype as a testing tool. We compared Lind against three existing virtualization systems – Docker, LXC, and Graphene. We chose these three systems because they currently represent the most widely-used VM design models for securing the OS kernel. LXC is a well-known container designed specifically for the Linux kernel. Docker is a widely-used container that wraps an application in a self-contained filesystem, while Graphene is an open source library OS designed to run an application in a virtual machine environment. Lastly, we also tested Native Linux to serve as a baseline for comparison. Our tests were designed to answer four fundamental questions:

How does Lind compare to other virtualization systems in protecting against zero-day Linux kernel bugs? (Section 5.1)

How much of the underlying kernel code is exposed, and is thus vulnerable in different virtualization systems? (Section 5.2)

If Lind's SafePOSIX construction has bugs, how severe an impact would this vulnerability have? (Section 5.3)

In the Lind prototype, what would be the expected performance overhead in real-world applications? Can developers make use of the "popular paths" metric to develop practical systems? (Section 5.4)

5.1 Linux Kernel Bug Test and Evaluation

Setup. To evaluate how well each virtualization system protects the Linux kernel against reported zero-day bugs, we examined a list of 69 historical bugs that had been identified and patched in versions 3.13.0 and 3.14.1 of the Linux kernel [13]. By consulting the National Vulnerability Database (NVD) [31], we obtained a list of all CVEs [11] that were known to exist in these Linux kernel versions as of September 2015; we found 69 such vulnerabilities. By analyzing security patches for those bugs, we were able to identify the lines of code in the kernel that correspond to each one.

In the following evaluation, we assume that a bug is potentially triggerable if the lines of code that were changed in the patch are reached (i.e., the same metric described in Section 3). This measure may overestimate potential danger posed by a system since simply reaching the buggy code does not mean that guest code actually has enough control to exploit the bug. However, this overestimate should apply equally to all of the systems we tested, which means it is still a useful method of comparison.

Next, we sought out proof-of-concept code that could trigger each bug. We were able to obtain or create code to trigger nine out of the 69 bugs [16]. For the rest, we used the Trinity system call fuzzer [42] on Linux 3.14.1 (referred to as "Native" Linux in Table 2). By comparing the code reached during fuzzing with the lines of code affected by security patches, we were able to identify an additional 26 bugs that could be triggered. All together, we identified a total of 35 bugs that we were able to trigger from user space, and these formed our final dataset for the evaluation.

We then evaluated the protection afforded by four virtualization systems (including Lind) by attempting to trigger the 35 bugs from inside each one. The host system for each test ran a version of Linux 3.14.1 with gcov instrumentation enabled. For the nine bugs that we could trigger directly, we ran the proof-of-concept exploit inside the guest. For the other 26, we ran the Trinity fuzzer inside the guest, exercising each system call 1,000,000 times with random inputs. Finally, we checked whether the lines of code containing each bug were reached in the host kernel, indicating that the guest could have triggered the bug.

Results. We found that a substantial number of bugs could be triggered in existing virtualization systems, as shown in Table 2. All (100%) bugs were triggered in Native Linux, while the other programs had lower rates:

8/35 (22.9%) in Docker, 12/35 (34.3%) in LXC, and 8/35 (22.9%) bugs in Graphene. Only 1 out of 35 bugs (2.9%) was triggered in Lind.

When we take a closer look at the results, we can see that these outcomes have a lot to do with the design principles of the virtualization systems and the way in which they handle system call requests. Graphene [43] is a library OS that relies heavily on the Linux kernel to handle system calls. Graphene's Linux library implements the Linux system calls using a variant of the Drawbridge [36] ABI, which has 43 functions. Those ABI functions are provided by the Platform Adaptation Layer (PAL), implemented using 50 calls to the kernel. It turns out that 8 vulnerabilities in our test were triggered by PAL's 50 system calls. By contrast, Lind only relies on 33 system calls, which significantly reduces risk and avoids 7 out of the 8 bugs.

Graphene supports many complex and risky system calls, such as `execve`, `msgsnd`, and `futex`, that reached the risky (unpopular) portion of the kernel and eventually led to kernel bugs. In addition, for many basic and frequently-used system calls like `open` and `read`, Graphene allows rarely-used flags and arguments to be passed down to the kernel, which triggered bugs in the unpopular paths. In Lind, all system calls only allow a restricted set of simple and frequently-used flags and arguments. One example from our test result is that Graphene allows `O_TMPFILE` flag to be passed to the `path_openat()` system call. This reached risky lines of code inside `fs/namei.c` in the kernel, and eventually triggered bug CVE-2015-5706. The same bug was triggered in the same way inside Docker and LXC, but was successfully prevented by Lind, due to its strict control of flags and arguments. In fact, the design of Graphene requires extensive interaction with the host kernel and, hence, has many risks. The developers of Graphene manually conducted an analysis of 291 Linux vulnerabilities from 2011 to 2013, and found out that Graphene's design can not prevent 144 of those vulnerabilities.

LXC [28] is an operating-system-level virtualization container that uses Linux kernel features to achieve containment. Docker [15] is a Linux container that runs on top of LXC. The two containers have very similar design features that both rely directly on the Linux kernel to handle system call requests. Since system calls inside Docker are passed down to LXC and then into the kernel, we found out that all 8 kernel vulnerabilities triggered inside Docker were also triggered with LXC. In addition, LXC interacts with the kernel via its `liblxc` library component, which triggered the extra 4 bugs.

It should be noted that although the design of Lind only accesses popular paths in the kernel and implements SafePOSIX inside of a sandbox, there are a few fundamental building blocks for which Lind must rely on the kernel. For example, `mmap` and `threads` cannot be recreated inside SafePOSIX without interaction with the kernel, since there have to be some basic operations to access the hardware. Therefore, Lind passes `mmap` and `threads` directly to the kernel, and any vulnerabilities related to them are unavoidable. CVE-2014-4171 is a bug triggered by `mmap` inside Lind. It was also triggered inside Docker, LXC, and Graphene, indicating that those systems rely on the kernel to perform `mmap` operations as well.

Our initial results suggest that bugs are usually triggered by extensive interaction with the unpopular paths in the kernel through complex system calls, or basic system calls with complicated or rarely used flags. The *Lock-in-Pop* design, and thus Lind, provides strictly controlled access to the kernel, and so poses the least risk.

Vulnerability	Native Linux	Docker	LXC	Graphene	Lind
CVE-2015-5706	✓	✓	✓	✓	✗
CVE-2015-0239	✓	✗	✓	✗	✗
CVE-2014-9584	✓	✗	✗	✗	✗
CVE-2014-9529	✓	✗	✓	✗	✗
CVE-2014-9322	✓	✓	✓	✓	✗
CVE-2014-9090	✓	✗	✗	✗	✗
CVE-2014-8989	✓	✓	✓	✓	✗
CVE-2014-8559	✓	✗	✗	✗	✗
CVE-2014-8369	✓	✗	✗	✗	✗
CVE-2014-8160	✓	✗	✓	✗	✗
CVE-2014-8134	✓	✗	✓	✓	✗
CVE-2014-8133	✓	✗	✗	✗	✗
CVE-2014-8086	✓	✓	✓	✗	✗
CVE-2014-7975	✓	✗	✗	✗	✗
CVE-2014-7970	✓	✗	✗	✗	✗
CVE-2014-7842	✓	✗	✗	✗	✗
CVE-2014-7826	✓	✗	✗	✓	✗
CVE-2014-7825	✓	✗	✗	✓	✗
CVE-2014-7283	✓	✗	✗	✗	✗
CVE-2014-5207	✓	✗	✗	✗	✗
CVE-2014-5206	✓	✓	✓	✗	✗
CVE-2014-5045	✓	✗	✗	✗	✗
CVE-2014-4943	✓	✗	✗	✗	✗
CVE-2014-4667	✓	✗	✗	✓	✗
CVE-2014-4508	✓	✗	✗	✗	✗
CVE-2014-4171	✓	✓	✓	✓	✓
CVE-2014-4157	✓	✗	✗	✗	✗
CVE-2014-4014	✓	✓	✓	✗	✗
CVE-2014-3940	✓	✓	✓	✗	✗
CVE-2014-3917	✓	✗	✗	✗	✗
CVE-2014-3153	✓	✗	✗	✗	✗
CVE-2014-3144	✓	✗	✗	✗	✗
CVE-2014-3122	✓	✗	✗	✗	✗
CVE-2014-2851	✓	✗	✗	✗	✗
CVE-2014-0206	✓	✗	✗	✗	✗
Vulnerabilities Triggered	**35/35 (100%)**	**8/35 (22.9%)**	**12/35 (34.3%)**	**8/35 (22.9%)**	**1/35 (2.9%)**

Table 2: Linux kernel bugs, and vulnerabilities in different virtualization systems (✓: vulnerability triggered; ✗: vulnerability not triggered).

5.2 Comparison of Kernel Code Exposure in Different Virtualization Systems

Setup. To determine how much of the underlying kernel can be executed and exposed in each system, we conducted system call fuzzing with Trinity (similar to our approach in Section 3) to obtain kernel traces. This helps us understand the potential risks a virtualization

Virtualization system	# of Bugs	Kernel trace (LOC)		
		Total coverage	In popular paths	In risky paths
LXC	12	127.3K	70.9K	56.4K
Docker	8	119.0K	69.5K	49.5K
Graphene	8	95.5K	62.2K	33.3K
Lind	1	70.3K	70.3K	0

Table 3: Reachable kernel trace analysis for different virtualization systems.

system may pose based upon how much access it allows to the kernel code. All experiments were conducted under Linux kernel 3.14.1.

Results. We obtained the total reachable kernel trace for each tested system, and further analyzed the components of those traces. These results, shown in Table 3, affirm that Lind accessed the least amount of code in the OS kernel. More importantly, all the kernel code it did access was in the popular kernel paths, which contain fewer bugs (Section 3.2). A large portion of the kernel paths accessed by Lind lie in `fs/` and perform file system operations. To restrict file system calls to popular paths, Lind allows only basic calls, like `open()`, `close()`, `read()`, `write()`, `mkdir()`, and `rmdir()`, and permits only commonly-used flags like `O_CREAT`, `O_EXCL`, `O_APPEND`, `O_TRUNC`, `O_RDONLY`, `O_WRONLY`, and `O_RDWR` for `open()`.

The other virtualization systems all accessed a substantial number of code paths in the kernel, and they all accessed a larger section from the unpopular paths. This is because they rely on the underlying host kernel to implement complex functionality. Therefore, they are more dependent on complex system calls, and allow extensive use of complicated flags. For example, Graphene's system call API supports multiple processes via `fork()` and signals, and therefore accesses many risky lines of code. For basic and frequently-used system calls like `open`, Graphene allows rarely-used flags, such as `O_TMPFILE` and `O_NONBLOCK` to pass down to the kernel, thus reaching risky lines in the kernel that could lead to bugs. By default, Docker and LXC do not wrap or filter system calls made by applications running in a container. Thus, programs have access to basically all the system calls, and rarely used flags, such as `O_TMPFILE`, `O_NONBLOCK`, and `O_DSYNC`. Again, this means they can reach risky lines of code in the kernel.

To summarize, our analysis suggests that Lind triggers the fewest kernel bugs because it has better control over the portions of the OS kernel accessed by applications.

5.3 Impact of Potential Vulnerabilities in Lind's SafePOSIX Re-creation

Setup. To understand the potential security risks if Lind's SafePOSIX re-creation has vulnerabilities, we conducted system call fuzzing with Trinity to obtain the reachable kernel trace in Linux kernel 3.14.1. The goal is to see how much of the kernel is exposed to SafePOSIX. Since our SafePOSIX runs inside the Repy sandbox kernel, fuzzing it suffices to determine the portion of the kernel reachable from inside the sandbox.

Virtualization system	# of Bugs	Kernel trace (LOC)		
		Total coverage	In popular paths	In risky paths
Lind	1	70.3K	70.3K	0
Repy	1	74.4K	74.4K	0

Table 4: Reachable kernel trace analysis for Repy.

Results. The results are shown in Table 4. The trace of Repy is slightly larger (5.8%) than that of Lind. This larger design does not allow attackers or bugs to access the risky paths in the OS kernel, and it leaves open only a small number of additional popular paths. These are added because some functions in Repy have more capabilities for message sending and network connection than Lind's system call interface. For example, in Repy, the `sendmessage()` and `openconnection()` functions could reach more lines of code when fuzzed. However, the kernel trace of Repy still lies completely within the popular paths that contain fewer kernel bugs. Thus, the Repy sandbox kernel has only a very slim chance of triggering OS kernel bugs.

Since it is the direct point of contact with the OS kernel, in theory, the Repy sandbox kernel could be a weakness in the overall security coverage provided by Lind. Nevertheless, the results above show that, even if it has a bug or failure, the Repy kernel should not substantially increase the risk of triggering bugs.

5.4 Practicality Evaluation

The purpose of our practicality evaluation is to show that the "popular paths" metric is practical in building real-world systems. Overhead is expected. We have not optimized our Lind prototype to try to improve performance, since that is not our main purpose for building the prototype.

Setup. We ran a few programs of different types to understand Lind's performance impact. All applications ran unaltered and correctly in Lind. To run the applications, it was sufficient to just recompile the unmodified source code using NaCl's compiler and Lind's `glibc` to call into SafePOSIX.

To measure Lind's runtime performance overhead compared to Native Linux when running real-world applications, we first compiled and ran six widely-used legacy applications: a prime number calculator Primes 1.0, GNU Grep 2.9, GNU Wget 1.13, GNU Coreutils 8.9, GNU Netcat 0.7.1, and K&R Cat. We also ran more extensive benchmarks on two large legacy applications, Tor 0.2.3 and Apache 2.0.64, in Lind. We used Tor's built-in benchmark program and Apache's benchmarking tool `ab` to perform basic testing operations and record the execu-

Application	Native Code	Lind	Impact
Primes	10000 ms	10600 ms	1.06x
GNU Grep	65 ms	260 ms	4.00x
GNU Wget	25 ms	96 ms	3.84x
GNU Coreutils	275 ms	920 ms	3.35x
GNU Netcat	780 ms	2180 ms	2.79x
K&R Cat	20 ms	125 ms	6.25x

Table 5: Execution time performance results for six real-world applications: Native Linux vs. Lind.

Benchmark	Native Code	Lind	Impact
Digest Tests:			
Set	54.80 nsec/element	176.86 nsec/element	3.22x
Get	42.30 nsec/element	134.38 nsec/element	3.17x
Add	11.69 nsec/element	53.91 nsec/element	4.61x
IsIn	8.24 nsec/element	39.82 nsec/element	4.83x
AES Tests:			
1 Byte	14.83 nsec/B	36.93 nsec/B	2.49x
16 Byte	7.45 nsec/B	16.95 nsec/B	2.28x
1024 Byte	6.91 nsec/B	15.42 nsec/B	2.23x
4096 Byte	6.96 nsec/B	15.35 nsec/B	2.21x
8192 Byte	6.94 nsec/B	15.47 nsec/B	2.23x
Cell Sized	6.81 nsec/B	14.71 nsec/B	2.16x
Cell Processing:			
Inbound	3378.18 nsec/cell	8418.03 nsec/cell	2.49x
(per Byte)	6.64 nsec/B	16.54 nsec/B	-
Outbound	3384.01 nsec/cell	8127.42 nsec/cell	2.40x
(per Byte)	6.65 nsec/B	15.97 nsec/B	-

Table 6: Performance results on Tor's built-in benchmark program: Native Linux vs. Lind.

tion time.

Results. Table 5 shows the runtime performance for the six real-world applications mentioned above. The Primes application run in Lind has a 6% performance overhead. The small amount of overhead is generated by NaCl's instruction alignment at build time. We expect other CPU bound processes to behave similarly.

The other five applications require repeated calls into SafePOSIX, and this additional computation produced the extra overhead.

A summary of the results for Tor is shown in Table 6. The benchmarks focus on cryptographic operations, which are CPU intensive, but they also make system calls like `getpid` and reads to `/dev/urandom`. The digest operations time the access of a map of message digests. The AES operations time includes encryptions of several sizes and the creation of message digests. Cell processing executes full packet encryption and decryption. In our test, Lind slowed down these operations by 2.5x to 5x. We believe these slowdowns are due to the increased code size produced by NaCl, and the increased overhead from Lind's SafePOSIX system call interface.

Results for the Apache benchmarking tool `ab` are presented in Table 7. In the set of experiments, Lind produced performance slowdowns around 2.7x. Most of the overhead was incurred due to system call operations inside the SafePOSIX re-creation.

Performance overhead in Lind is reasonable, considering that we did not specifically optimize any part of the code to improve speed. It should also be noted that

# of Requests	Native Code	Lind	Impact
10	900 ms	2400 ms	2.67x
20	1700 ms	4700 ms	2.76x
50	4600 ms	13000 ms	2.83x
100	10000 ms	27000 ms	2.70x

Table 7: Performance results on Apache benchmarking tool `ab`: Native Linux vs. Lind.

performance slowdown is common in virtualization systems. For example, Graphene [43] also shows an overhead ranging from 1.4x to 2x when running applications such as the Apache web server and the Unixbench suite [44]. In many cases, Lind shares the same magnitude of slowdown with Graphene. Lind's ability to run a variety of programs demonstrates the practicality of our "popular paths" metric.

6 Limitations

One of our challenges in conducting this study was deciding where to place the limits of its scope. To explore any one strategy in depth, we felt it was necessary to intentionally exclude consideration of a few other valid approaches. These choices may have placed some limitations on our results.

One limitation is that there are some types of bugs that are difficult to evaluate using our metric. For example, bugs caused by a race condition, or that involve defects in internal kernel data structures, or that require complex triggering conditions across multiple kernel paths, may not be immediately identified using our metric. As we continue to refine our metric, we will also look to evolve our evaluation criteria to find and protect against more complex types of bugs.

Another limitation is that our current metric concludes that certain lines of code in the kernel were reached or not. Though this is an important factor in exploiting a bug, it may not be fully sufficient for all bugs. While a stronger conclusion about bug exploitation conditions would be ideal, it would be hard to do so using a quantitative metric. Instead, it would require a more complicated manual process, which was outside the scope of this study.

7 Related Work

This section summarizes a number of earlier initiatives to ensure the safety of privileged code. The literature referenced in this section includes past efforts to design and build virtualized systems, as well as background information on technologies incorporated into Lind.

Lind incorporates a number of existing virtualization techniques, which are described below.

System Call Interposition (SCI) tracks all the system calls of processes such that each call can be modified or denied. Goldberg, et al. developed Janus [20, 48], which adopted a user-level "monitor" to filter system call requests based on user-specified policies. Garfinkel, et al.

proposed a delegating architecture for secure system call interposition called Ostia [18]. Their system introduced emulation libraries in the user space to mediate sensitive system calls issued by the sandboxed process. SCI is similar to the Lind isolation mechanism. However, SCI-based tools can easily be circumvented if the implementation is not careful [41].

Software Fault Isolation (SFI) transforms a given program so that it can be guaranteed to satisfy a security policy. Wahbe, et al. [49] presented a software approach to implementing fault isolation within a single address space. Yee, et al. from Google developed Native Client (NaCl) [51], an SFI system for the Chrome browser that allows native executable code to run directly in a browser. As discussed in Section 5, Lind adopts NaCl as a key component to ensure secure execution of binary code.

Language-based virtualization. Programming languages like Java, JavaScript, Lua [27], and Silverlight [40] can provide safety in virtual systems by "translating" application commands into a native language. Though many sandboxes implement the bulk of standard libraries in memory-safe languages like Java or C#, flaws in this code can still pose a threat [21, 34]. Any bug or failure in a programming language virtual machine is usually fatal. In contrast, the main component of Lind is built using Repy, which is a programming language with a very small TCB, minimizing the chance of contact with kernel flaws.

OS virtualization techniques include bare-metal hardware virtualization, such as VMware ESX Server, Xen [4], and Hyper-V, container systems such as LXC [28], BSD's jail, and Solaris zones, and hosted hypervisor virtualization, such as VMware Workstation, VMware Server, VirtualPC and VirtualBox. Security by isolation [2, 9, 23, 50] uses containment to provide safe executing environments for multiple user-level virtual environments sharing the same hardware. However, this approach is limited due to the large attack surface exposed by most hypervisors.

Library OSes allow applications to efficiently gain the benefits of virtual machines by refactoring a traditional OS kernel into an application library. Porter, et al. developed Drawbridge [36], a library OS that presents a Windows persona for Windows applications. Similar to Lind, it restricts access from usermode to the host OS through operations that pass through the security monitor. Baumann, et al. presented Bascule [5], an architecture for library OS extensions based on Drawbridge that allows application behavior to be customized by extensions loaded at runtime. The same team also developed Haven [6], which uses a library OS to implement shielded execution of unmodified server applications in an untrusted cloud host. Tsai, et al. developed Graphene [43], a library OS that executes both single and multi-process applications with low performance overhead.

The key distinction between Lind and other existing library OSes is that Lind leverages our "popular paths" metric to verify that it only accesses the safer part of the kernel. Existing library OSes trust the underlying host kernel to perform many functions, and filter only certain system calls. Our work and previous library OSes are orthogonal, but we provide useful insights with our "popular paths" metric.

8 Conclusion

In this paper, we proposed a new security metric based on quantitative measures of kernel code execution when running user applications. Our metric evaluates if the lines of kernel code executed have the potential to trigger zero-day bugs. Our key discovery is that popular kernel paths contain significantly fewer bugs than other paths. Based on this insight, we devise a new design for a secure virtual machine called *Lock-in-Pop*. As the name implies, the design scheme locks away access to all kernel code except that found in paths frequently used by popular programs. We test the *Lock-in-Pop* idea by implementing a prototype virtual machine called Lind, which features a minimized TCB and prevents direct access to application calls from less-used, riskier paths. Instead, Lind supports complex system calls by securely re-creating essential OS functionality inside a sandbox. In tests against Docker, LXC, and Graphene, Lind emerged as the most effective system in preventing zero-day Linux kernel bugs.

So that other researchers may replicate our results, we make all of the kernel trace data, benchmark data, and source code for this paper available [24].

Acknowledgements

We thank our shepherd, Dan Williams, and the anonymous reviewers for their valuable comments. We would also like to thank Lois Anne DeLong for her efforts on this paper, as well as Chris Matthews, Shengqian Ji, Qishen Li, Ali Gholami, Wenzheng Xu, and Yanyan Zhuang for their contributions to this project. Our work on *Lock-in-Pop* was supported by U.S. National Science Foundation Award 1223588.

References

[1] Debian Popularity Contest. http://popcon.debian.org/main/index.html. Accessed December 2014.

[2] Qubes OS. http://www.qubes-os.org. Accessed November 2015.

[3] Intel Virtualization Technology Specification for the Intel Itanium Architecture (VT-i), April 2005.

[4] BARHAM, P., DRAGOVICH, 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 SOSP'03* (2003), pp. 164–177.

[5] 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 Eurosys'13* (2013).

[6] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with haven. In *Proceedings of the OSDI'14* (2014).

[7] Berkeley software distribution. http://www.bsd.org. Accessed September 2016.

[8] CAPPOS, J., DADGAR, A., RASLEY, J., SAMUEL, J., BESCHASTNIKH, I., BARSAN, C., KRISHNAMURTHY, A., AND ANDERSON, T. Retaining sandbox containment despite bugs in privileged memory-safe code. In *Proceedings of the CCS'10* (2010).

[9] CHEN, X., GARFINKEL, T., LEWIS, E. C., SUBRAHMANYAM, P., WALDSPURGER, C. A., BONEH, D., DWOSKIN, J., AND PORTS, D. R. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. *SIGPLAN Not. 43*, 3 (Mar. 2008), 2–13.

[10] CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. *An empirical study of operating systems errors*, vol. 35. ACM, 2001.

[11] Common Vulnerabilities and Exposures. https://cve.mitre.org.

[12] CVE-2016-5195. Dirty COW - (CVE-2016-5195) - Docker Container Escape. `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-5195`, 2016.

[13] CVE Details Datasource. http://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/version_id-163187/Linux-Linux-Kernel-3.14.1.html. Accessed October 2014.

[14] CVE DETAILS. 14 CVE Docker Vulnerabilities Reported. http://www.cvedetails.com/product/28125/Docker-Docker.html?vendor_id=13534, 2017.

[15] Docker. https://www.docker.com. Accessed September 2016.

[16] Exploit Database. https://www.exploit-db.com. Accessed October 2014.

[17] FBI Tweaks Stance On Encryption BackDoors, Admits To Using 0-Day Exploits. http://www.darkreading.com/endpoint/fbi-tweaks-stance-on-encryption-backdoors-admits-to-using-0-day-exploits/d/d-id/1323526.

[18] GARFINKEL, T., PFAFF, B., AND ROSENBLUM, M. Ostia: A delegating architecture for secure system call interposition. In *Proceedings of the NDSS'04* (2004).

[19] gcov(1) - Linux man page. http://linux.die.net/man/1/gcov. Accessed October 2014.

[20] GOLDBERG, I., WAGNER, D., THOMAS, R., AND BREWER, E. A secure environment for untrusted helper applications (confining the wily hacker). In *Proceedings of the USENIX UNIX Security Symposium'96* (1996).

[21] Learn about java technology. http://www.java.com/en/about/.

[22] KELLER, E., SZEFER, J., REXFORD, J., AND LEE, R. B. Nohype: virtualized cloud infrastructure without the virtualization. In *ACM SIGARCH Computer Architecture News* (2010), vol. 38, ACM, pp. 350–361.

[23] LI, C., RAGHUNATHAN, A., AND JHA, N. Secure virtual machine execution under an untrusted management os. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on* (July 2010), pp. 172–179.

[24] Lind, a new generation of secure virtualization. https://lind.poly.edu/.

[25] Linux kernel zero-day flaw puts 'tens of millions' of PCs, servers and Android devices at risk. http://www.v3.co.uk/v3-uk/news/2442582/linux-kernal-zero-day-flaw-puts-tens-of-millions-of-pcs-servers-and-android-devices-at-risk.

[26] Linux Test Project. https://linux-test-project.github.io/. Accessed February 2015.

[27] The programming language Lua. www.lua.org. Accessed October 2015.

[28] Linux Container (LXC). https://linuxcontainers.org. Accessed September 2016.

[29] MAYER, A., AND SYKES, A. A probability model for analysing complexity metrics data. *Software Engineering Journal 4*, 5 (1989), 254–258.

[30] NSA Discloses 91 Percent Of Vulns It Finds, But How Quickly? http://www.darkreading.com/vulnerabilities—threats/nsa-discloses-91-percent-of-vulns-it-finds-but-how-quickly/d/d-id/1323077.

[31] National Vulnerability Database. https://nvd.nist.gov/. Accessed September 2015.

[32] OZMENT, A., AND SCHECHTER, S. E. Milk or wine: does software security improve with age? In *Usenix Security* (2006).

[33] PALIX, N., THOMAS, G., SAHA, S., CALVÈS, C., LAWALL, J., AND MULLER, G. Faults in linux: ten years later. In *ACM SIGARCH Computer Architecture News* (2011), vol. 39, ACM, pp. 305–318.

[34] PAUL, N., AND EVANS., D. Comparing java and .net security: Lessons learned and missed. In *Computers and Security* (2006), pp. 338–350.

[35] Poisson Distribution. https://en.wikipedia.org/wiki/Poisson_distribution.

[36] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. C. Rethinking the library os from the top down. In *Proceedings of the ASPLOS'11* (Newport Beach, California, USA, 2011), pp. 291–304.

[37] Qemu. http://wiki.qemu.org/Main_Page. Accessed September 2016.

[38] Seattle's Repy V2 Library. https://seattle.poly.edu/wiki/RepyV2API. Accessed September 2014.

[39] SHIUE, W.-K., AND BAIN, L. J. Experiment size and power comparisons for two-sample poisson tests. *Applied Statistics* (1982), 130–134.

[40] Microsoft Silverlight. http://www.microsoft.com/silverlight/. Accessed October 2015.

[41] TAL GARFINKEL. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools.

[42] Trinity, a Linux System call fuzz tester. http://codemonkey.org.uk/projects/trinity/. Accessed November 2014.

[43] 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 EuroSys'14* (Amsterdam, Netherlands, 2014).

[44] Unixbench. https://github.com/kdlucas/byte-unixbench. Accessed September 2016.

[45] Virtualbox. https://www.virtualbox.org. Accessed September 2016.

[46] Vmware server. https://my.vmware.com/web/vmware/info?slug=infrastructure_operations_management/vmware_server/2_0.

[47] Vmware workstation. https://www.vmware.com/products/workstation. Accessed September 2016.

[48] WAGNER, D. A. Janus: An approach for confinement of untrusted applications. In *Tech. Rep. CSD-99-1056, University of California, Berkeley* (1999).

[49] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *SIGOPS Oper. Syst. Rev. 27, 5* (1993), pp. 203–216.

[50] YANG, J., AND SHIN, K. G. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2008), VEE '08, ACM, pp. 71–80.

[51] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the IEEE Symposium on Security and Privacy* (Berkeley, CA, USA, 2009), pp. 79–93.

[52] 0-day exploits more than double as attackers prevail in security arms race. http://arstechnica.com/security/2016/04/0-day-exploits-more-than-double-as-attackers-prevail-in-security-arms-race/. Accessed September 2016.

Fast and Precise Retrieval of Forward and Back Porting Information for Linux Device Drivers

Julia Lawall, Derek Palinski, Lukas Gnirke, Gilles Muller
Sorbonne Universités/UPMC/Inria/LIP6

Abstract

Porting Linux device drivers to target more recent and older Linux kernel versions to compensate for the ever-changing kernel interface is a continual problem for Linux device driver developers. Acquiring information about interface changes is a necessary, but tedious and error prone, part of this task. In this paper, we propose two tools, *Prequel* and *gcc-reduce*, to help the developer collect the needed information. Prequel provides language support for querying git commit histories, while gcc-reduce translates error messages produced by compiling a driver with a target kernel into appropriate Prequel queries. We have used our approach in porting 33 device driver files over up to 3 years of Linux kernel history, amounting to hundreds of thousands of commits. In these experiments, for 3/4 of the porting issues, our approach highlighted commits that enabled solving the porting task. For many porting issues, our approach retrieves relevant commits in 30 seconds or less.

1 Introduction

The Linux kernel evolves rapidly, with around 70,000 non-merge commits accepted per year since 2013. Commits may fix bugs and add new functionalities, but may also change the interfaces between the kernel core and services that run at the kernel level. For example, between Linux 3.8 (February 2013) and Linux 4.9 (December 2016), 2,439 of the 19,473 functions exported to kernel modules were dropped and 10,056 new exported functions were introduced. This rate of interface changes allows the Linux kernel to rapidly address new needs and resolve performance and security bugs.

While the fast rate of change of Linux kernel interfaces has benefits, it poses challenges for developers of services, such as device drivers, that rely on the kernel interface. Such a developer has to target a particular version of the Linux kernel, but any version chosen will be quickly out of date. Furthermore, potential users of the device may rely on earlier kernel versions, due to *e.g.*, local customizations or stability requirements. The tight dependence of device drivers on fast-changing kernel interfaces means that there is a continual need for forward porting driver code to the interfaces supported by more recent kernel versions, and back porting driver code to the interfaces supported by older kernel versions. This requires a lot of effort for device manufacturers who want to support the needs of a range of clients and for device users who rely on specific kernels.

A major challenge in forward or back porting a device driver is to find out where changes are needed and what changes should be performed. Many drivers interact with the kernel interface in similar ways, and thus change examples are likely to be available in the code history. Still, finding these examples effectively requires knowing what to look for. One approach is to compile the driver with the target kernel and take the resulting error and warning messages as a starting point for identifying porting issues. These messages, however, may be redundant, if one error causes the compiler to misinterpret other code, and may be too concise to sufficiently characterize a porting problem. Even when it is possible to pinpoint the porting issues, an even greater challenge is to find relevant examples among the hundreds of commits per day to the Linux kernel. Git,[1] used for change management in the Linux kernel, supports search for a single regular expression within individual changed lines, via the commands `git log -G` and `git log -S`. But particular terms may appear within changed lines for many reasons, not all of which relate to porting issues, and thus git often returns many irrelevant commits.

The difficulty of obtaining relevant information on how to port a driver thus calls for tool support. In this paper, we propose an approach to ease driver porting based on two tools: *Prequel* and *gcc-reduce*. *Prequel* searches

[1] https://git-scm.com/

in a git commit history for commits matching a query. Queries can include constraints on both changed and unchanged lines, allowing Prequel to obtain more precise results than git. Prequel furthermore ranks the resulting commits according to the degree of success of the match, rather than chronologically as done by git, giving the driver maintainer easy access to the most relevant results. *Gcc-reduce* complements Prequel by creating a bridge from the compiler. Given a set of compiler errors and warnings, gcc-reduce reduces them to those that are relevant to porting, and collects complementary information from the source code. Gcc-reduce then generates Prequel queries based on the collected information. Although the possibility remains to write Prequel queries by hand, gcc-reduce is able to generate most queries relevant to driver porting automatically. Overall, our approach permits the developer to save time and effort, by obtaining change examples that are relevant to the porting problem.

The contributions of this paper are as follows:

- Via a case study, we identify two key challenges in driver porting: determining 1) where changes are needed and 2) what changes should be performed.
- We propose the tools Prequel and gcc-reduce that automatically collect information to address these challenges.
- We evaluate Prequel and gcc-reduce by porting 33 device driver files introduced into the Linux kernel in 2013 or 2015 to or from Linux 4.6, released in May 2016. Our approach provides information from the git commit history that enables us to carry out the port for 3/4 of the issues encountered.
- We show that our approach is suitable for use on a standard laptop, with many patch queries completing in 30 seconds or less.
- We compare our approach to the use of git to query the commit history and the use of Google to find relevant change suggestions. For queries such as field type changes, we find that git returns many irrelevant commits. For only 33% of the issues does Google return possibly relevant results among the top 6 entries in the query summary page. In contrast, for our ported 33 driver files, the top ranked commit returned by our approach is helpful for 86% of the porting issues.

2 Motivating Example

To understand the challenges in obtaining adequate information on how to carry out a driver port, we consider the lms501kf03 TFT LCD panel driver, introduced into the Linux kernel in February 2013 in the commit 1be9ca2, and first released in Linux 3.9. The driver consists of a single `.c` file, drivers/video/backlight/lms501kf03.c. We forward port this code over 16 Linux kernel releases, to Linux 4.6, released in May 2016.

The experiment. Compiling the original driver with the Linux 4.6 kernel[2] produces the errors and warnings shown in Figure 1. There are two errors (lines 1 and 8), about the `suspend` and `resume` fields being unknown, and six warnings. The warnings appear to be triggered by the same cause as the errors, and thus we focus on the latter. Specifically, we need to find examples of how to remove `suspend` and `resume` fields, and then see what we can infer from those examples for porting our driver.

Focusing on `suspend`, we can try the following git command, considering commits between the kernel version originally targeted by the driver and the version that is the target of the port:

```
git log -p -G "\<suspend\>" --diff-filter=M 1be9ca2..v4.6
```

`-p` prints the changed lines, `-G "\<suspend\>"` restricts the results to commits that contain the word `suspend` on a changed line, and `--diff-filter=M` restricts the results to commits that perform modifications, as opposed to adding or removing files.

Despite the relative sophistication of this git command, many of the results are completely irrelevant. For example, the first result, commit ba41e1b, removes a reference to a `suspend` field and adds another such reference on the same structure. Such a commit does not help fix a reference to a field that no longer exists. Rather, we need commits in which `suspend` appears on removed lines, but not added ones, which is not expressible with `git log -G`. Subsequent commits give similar results. Indeed, these commits are modifying `suspend` fields in structures having types different from the one, `spi_driver`, used in our driver.

Ideally, we could extend the `git log -G` to include the name of the structure type, but this type name is not likely to be on the same line as the field reference. Nevertheless, we can use the search command of the git viewer to find occurrences of `spi_driver` in the context lines of the emitted patch code. This process might not succeed, because the type name can be arbitrarily distant from the changed lines. In our case, though, it is successful, but the user has to analyze and scroll over 7 occurrences of `spi_driver` within 569 commits before reaching a relevant commit 9d9f780, from January 2015.

Figure 2 shows extracts of the commit 9d9f780. This commit removes initializations of the `suspend` and `resume` fields, on lines 37-38, but also does many other

[2] `make drivers/video/backlight/lms501kf03.o`, with gcc (Ubuntu 4.8.4-2ubuntu1 14.04.3) 4.8.4.

```
drivers/video/backlight/lms501kf03.c:433:2: error: unknown field ‘suspend’ specified in initializer
  .suspend = lms501kf03_suspend,
  ^
drivers/video/backlight/lms501kf03.c:433:2: warning: missing braces around initializer [-Wmissing-braces]
drivers/video/backlight/lms501kf03.c:433:2: warning: (near initialization for ‘lms501kf03_driver.driver’) [-Wmissing-braces]
drivers/video/backlight/lms501kf03.c:433:2: warning: initialization from incompatible pointer type [enabled by default]
drivers/video/backlight/lms501kf03.c:433:2: warning: (near initialization for ‘lms501kf03_driver.driver.name’) [enabled by
      default]
drivers/video/backlight/lms501kf03.c:434:2: error: unknown field ‘resume’ specified in initializer
  .resume  = lms501kf03_resume,
  ^
drivers/video/backlight/lms501kf03.c:434:2: warning: excess elements in struct initializer [enabled by default]
drivers/video/backlight/lms501kf03.c:434:2: warning: (near initialization for ‘lms501kf03_driver’) [enabled by default]
```

Figure 1: Messages resulting from compiling the original Linux lms501kf03 TFT LCD panel driver with Linux 4.6

things. We thus next have to determine whether these other changes are relevant to our driver and whether other examples are needed. We focus on the changes in the definitions of the functions `as3935_suspend` and `as3935_resume` that are stored in the `suspend` and `resume` fields, respectively. The parameter lists of these functions are modified to change the type of the first parameter, and, in the case of the suspend function, to drop the second parameter. In the body of each function, a call to `spi_get_drvdata` on the original first parameter is replaced by a call to `dev_get_drvdata` on the new first parameter (lines 10-11 and 16-17). No change is made to compensate for dropping the second parameter of the suspend function, as it is unused.

We next compare the observed set of changes to the code found in the TFT LCD panel driver that we want to port. Figure 3 shows the relevant code fragments. Some code fragments are analogous to the ones modified in commit 9d9f780. For example, our driver also initializes `spi_driver` `suspend` and `resume` fields to locally defined functions, `lms501kf03_suspend` and `lms501kf03_resume`, respectively. These functions have the same list of parameters as found in commit 9d9f780, and the function stored in the `suspend` field again has the property that the second parameter is not used. The associated changes found in the sample commit can thus be applied directly.

The set of changes illustrated in commit 9d9f780 are, however, insufficient for determining how to update the function bodies. While the suspend and resume functions affected by commit 9d9f780 use their first parameter only in calls to `spi_get_drvdata`, the suspend and resume functions in our driver call `dev_get_drvdata` and `dev_dbg` on values derived from this parameter. Thus, we need more examples. Searching further through the commits, we find commit 01f9326 from March 2013 that is similar to the commit shown in Figure 2 but contains the following change in the functions stored in the `suspend` and `resume` fields:

```
-  struct snd_card *card = dev_get_drvdata(&spi->dev);
+  struct snd_card *card = dev_get_drvdata(dev);
```

```
diff --git a/drivers/iio/proximity/as3935.c ...
index 466aa43..bc0d68e 100644
--- a/drivers/iio/proximity/as3935.c
+++ b/drivers/iio/proximity/as3935.c
@@ -273,9 +273,9 @@ static void calibrate_as3935(...)
 #ifdef CONFIG_PM_SLEEP
-static int as3935_suspend(struct spi_device *spi,
-      pm_message_t msg)
+static int as3935_suspend(struct device *dev)
 {
-  struct iio_dev *indio_dev = spi_get_drvdata(spi);
+  struct iio_dev *indio_dev = dev_get_drvdata(dev);
@@ -293,9 +293,9 @@ err_suspend:
-static int as3935_resume(struct spi_device *spi)
+static int as3935_resume(struct device *dev)
 {
-  struct iio_dev *indio_dev = spi_get_drvdata(spi);
+  struct iio_dev *indio_dev = dev_get_drvdata(dev);
@@ -311,9 +311,12 @@ err_resume:
+
+static SIMPLE_DEV_PM_OPS(as3935_pm_ops, as3935_suspend,
+      as3935_resume);
+#define AS3935_PM_OPS (&as3935_pm_ops)
+
 #else
-#define as3935_suspend NULL
-#define as3935_resume  NULL
+#define AS3935_PM_OPS NULL
 #endif
@@ -441,12 +444,11 @@ static struct spi_driver as3935_driver
   .driver = {
     .name   = "as3935",
     .owner  = THIS_MODULE,
+    .pm     = AS3935_PM_OPS,
   },
   .probe    = as3935_probe,
   .remove   = as3935_remove,
   .id_table = as3935_id,
-  .suspend  = as3935_suspend,
-  .resume   = as3935_resume,
 };
```

Figure 2: Example update on suspend and resume. Some context lines are omitted for readability.

Commit eba3bfb from April 2013 illustrates the case of a `dev_dbg` call:

```
-  dev_dbg(&spi->dev, "lcd->power = %d\n", lcd->power);
+  dev_dbg(dev, "lcd->power = %d\n", lcd->power);
```

All these changes, from these different commits, provide a model for the porting of the TFT LCD panel driver.

```
#if defined(CONFIG_PM)
static int lms501kf03_suspend(struct spi_device *spi,
      pm_message_t mesg)
{
  struct lms501kf03 *lcd = dev_get_drvdata(&spi->dev);
  dev_dbg(&spi->dev, "lcd->power = %d\n", lcd->power);
  return lms501kf03_power(lcd, FB_BLANK_POWERDOWN);
}
static int lms501kf03_resume(struct spi_device *spi)
{
  struct lms501kf03 *lcd = dev_get_drvdata(&spi->dev);
  lcd->power = FB_BLANK_POWERDOWN;
  return lms501kf03_power(lcd, FB_BLANK_UNBLANK);
}
#else
#define lms501kf03_suspend NULL
#define lms501kf03_resume NULL
#endif

static struct spi_driver lms501kf03_driver = {
  .driver = {
    .name = "lms501kf03",
    .owner = THIS_MODULE,
  },
  .probe = lms501kf03_probe,
  .remove = lms501kf03_remove,
  .shutdown = lms501kf03_shutdown,
  .suspend = lms501kf03_suspend,
  .resume = lms501kf03_resume,
};
```

Figure 3: lms501kf03 TFT LCD panel driver extract

Assessment. Our example illustrates clearly that compiler error messages are helpful, but there can be a significant difference between the set of the errors raised by the compiler and the set of changes required. In Figure 1, we see that the compiler may report errors and warnings that are actually side-effects of other issues, and do not help to identify the set of changes required. Furthermore, many changes, such as the changes in the definitions of `lms501kf03_suspend` and `lms501kf03_resume`, are required but were not reported by the compiler, and thus adequate examples of changes on similar drivers are necessary to determine what changes are needed and how to carry them out. Then, finding even one commit that illustrates a specific problem is a major challenge. As no one commit may illustrate all of the issues relevant to a particular driver, repeating this process to find multiple commits may be necessary.

3 Prequel

The core of our approach is the process of searching for commits that illustrate how to perform a particular kind of change. Such a search must be able to take into account properties of both changed lines and the context in which the changed lines occur. For example, we would like to retrieve only commits that remove initializations of `suspend` and `resume` fields that are in `spi_driver` structures. To retrieve such commits, we propose a patch query language, PQL, that permits describing properties

```
1 @bad depends on after@
2 identifier i;
3 expression e;
4 @@
5 struct spi_driver i = {
6   .suspend
7     = e,
8 };

9 @rem depends on !bad@
10 identifier i;
11 expression e;
12 @@
13 struct spi_driver i = {
14 - .suspend
15     = e,
16 };
```

Figure 4: Patch query detecting removals of the `suspend` field from the initialization of a `spi_driver` structure

of both changed lines and their context and a tool Prequel that applies PQL queries to a git commit history. We first briefly present the syntax and semantics of PQL. More details are available in a technical report [5]. We then describe optimizations that allow using Prequel on a standard laptop.

3.1 PQL syntax and semantics

To describe changes and their context in systems code, we take inspiration from the program transformation tool Coccinelle [2, 8]. Coccinelle and its Semantic Patch Language (SmPL) offer a transformation language based on the familiar patch syntax. Coccinelle is widely used in Linux kernel development, and its notation is familiar to kernel developers. Our key insight is that a specification of which lines to add and remove can also be viewed as a description of the lines that have been added and removed, after the transformation has been performed. We thus propose a SmPL-like notation for PQL, providing a description, which we refer to as a *patch query*, of the effect of a previous transformation process. Prequel then applies a patch query to a series of commits.

Figure 4 shows a patch query that detects commits that remove an initialization of a `suspend` field in a `spi_driver` structure. The patch query consists of two rules, the rule `bad` on lines 1-8 followed by the rule `rem` on lines 9-16. We first focus on the latter. A patch query rule consists of a fragment of code that combines concrete terms, such as the type name `spi_driver`, with *metavariables*, declared at the top of the rule. `rem` uses metavariables for the name of the driver structure, `i` (line 13), which is declared to match any identifier (line 10), and the initial value of the `suspend` field `e` (line 15), which is declared to match any expression (line 11). Within the code fragment, `-` and `+` indicate tokens that must be removed or added by a matching commit, respectively. `rem` indicates that the token `suspend` must be removed (line 14).

The output of gcc indicates that we want to find commits that remove the `suspend` field, but does not indicate what other changes may be needed. For example, one way to remove a field is to rename it, in which case the initial value expression `e` may remain unchanged, while

another possibility is to remove both the field name and the initial value expression entirely. Prequel is designed based on the hypothesis that a user searches for a change to complete his understanding of that change, and thus Prequel provides approximate matching. Specifically, tokens that are annotated with - or + *must* be removed or added, respectively, but other tokens *may* be removed or added as well. To distinguish between more or less precise matches, Prequel returns the matching commits, ranked by the percentage of changed lines or hunks that contain an exact match of the specification.

The rule `rem` also does not guard against the possibility that the `suspend` field is simply moved around in the `spi_driver` structure, *i.e.*, removed but added back, or that one instance of a `suspend` field is removed but others remain. As gcc reports that `suspend` is unknown, we need to find example commits that remove the `suspend` field from the `spi_driver` completely. The rule `bad` (lines 1-8) extends the patch query to ensure this property. This rule matches a commit for which the state of the code after the commit, as indicated by `depends on after` on line 1, contains an initialization of a `suspend` field. Such commits are ones that we do *not* want to see. The rule `rem` then depends on the failure of the rule `bad`. Prequel returns only commits that match rules on which no other rules depend, and thus the only results are those that completely remove the `suspend` field (rule `rem`).

3.2 Optimizations

We want to support driver porting on a standard laptop, as would be most easily accessible to a kernel developer. The matching performed by Prequel, however, may be very expensive. Because a patch query may describe not only changed code, but also context code that occurs elsewhere in the same file, *e.g.*, the type `spi_driver` in our case, Prequel matches a patch query against complete files as they exist before and after a commit, and not just against the changed lines. Matching a query against all the files affected by hundreds of thousands of commits, as are found in several years of history of the Linux kernel, however, would be very time consuming. In practice, due to the diversity of the Linux kernel, for any given set of keywords, only a small percentage of the commits are likely to be relevant. Thus, performing such thorough matching is often unnecessary.

To reduce the set of commits considered in detail, Prequel first analyzes a patch query to identify keywords that must be present in or near the changes made by any commit that the patch query can match. For example, in the patch query of Figure 4, Prequel would select `suspend` as a keyword that must be present in the lines removed by a commit to allow a match. Prequel then searches for these keywords in the patch associated with each commit and ignores commits where they are not found, exploiting the fact that a patch is typically much smaller than the affected source files. Alternatively, for better performance, the user can prepare indices in advance, using the GNU utility ID Utils.[3] These indices map tokens to 1) the commits for which they occur on removed lines 2) the commits for which they occur on added lines, and 3) the commits for which they occur on changed lines or within 3 lines of context code (the default when using the `diff` command). Prequel uses the first two indices to identify commits that contain tokens annotated in the patch query with - or +, respectively, while it uses the third index to identify commits that contain unannotated tokens. Using an index is a choice left up to the user, because it trades flexibility in the range of considered commits for performance. Currently, the Prequel distribution includes indices for Linux versions 3.0 through 4.6, the range considered in our evaluation.

Limiting the set of commits considered based on keywords found in patches is only effective when the relevant keywords are found within or near the changed lines. Some relevant keywords, however, may appear far from any change. For example, in our case, if the `suspend` field was not dropped, but rather had its type changed, then the patch could be on the definition of the function stored in the `suspend` field, which could be far from any mention of `suspend`. When important keywords are expected to be far from changed lines, Prequel collects all of the files that contain the keywords in a reference version of the Linux kernel chosen by the user, *e.g.*, the source or target version of the port, and then considers only the commits that affect these files. It is also possible to provide an index mapping tokens to the reference version files in which they occur, to further improve performance.

The above strategies may incur false negatives: a keyword may appear as required in the code before or after the commit, but not close enough to changed lines, or a keyword may not appear anywhere in the reference version. To select commits, Prequel first tries keywords annotated with - or +, which must be within the changed lines, and only relies on unannotated keywords if consideration of the annotated keywords does not sufficiently reduce the number of commits to analyze in detail.

4 Gcc-reduce

Extracting the relevant information from compiler errors and from the source code to create Prequel patch queries is tedious and error prone. Our tool gcc-reduce provides a front end to Prequel for driver porting that automates this task. Because the kinds of error messages that the

[3] https://www.gnu.org/software/idutils/

compiler generates are limited, gcc-reduce can also automate the construction of patch queries, in most cases.

Gcc-reduce collects information required for the search for change examples from the errors and warnings[4] produced by compiling the driver with the target version and from the driver source code. Gcc-reduce expects the use of gcc for compilation; LLVM is known to give better error messages, but the support for compiling the kernel with LLVM is incomplete, and appears to be not well maintained.[5] Gcc-reduce then 1) reduces the resulting compiler errors to those that indicate porting issues, and 2) generates PQL patch queries from the information collected from the compiler error messages, as described below.

4.1 Error message reduction

As illustrated in Figure 1, gcc often gives multiple error messages that actually derive from the same problem. Three issues arise: 1) a problem recurs, 2) dataflow relationships imply that a problem in one part of the code makes another part of the code invalid, 3) the same as the second case, but triggered by structural relationships rather than dataflow.

To characterize the compiler errors, we have created a number of *error categories*, such as "unknown field error", illustrated by lines 1-3 of Figure 1. First, to detect recurring errors, for each error, gcc-reduce selects the corresponding error category and collects keywords that uniquely identify the problem. The keywords can come from the error message itself or from the source code. For example, for the error on lines 1-3 of Figure 1, the keywords are `suspend`, obtained from the error message and representing the affected field, and the name of the type of the enclosing structure, `spi_driver`, which is obtained from the source code. `gcc-reduce` discards subsequent errors of the same error category with the same keywords.

Second, analogous to the notion of *gen* and *kill* sets in dataflow analysis [1], gcc-reduce collects for each error an *input* set, containing keywords that, if they have been associated with a previous error, imply that the current error is redundant, and an *impact* set, containing keywords that, if they are associated with a future error, imply that the future error is redundant. In our example, the input set is {`struct spi_driver.suspend`, `struct spi_driver`}; if some other error has been reported related to the `suspend` field of a `spi_driver` structure, then resolving that error is likely to also resolve the one in our example, and the error in our example is not needed. Furthermore, if the entire `spi_driver` structure has been found to be invalid, then there is no need for an error about one of its fields. The impact set then is {`struct spi_driver.suspend`}, indicating that a problem has been found with the `suspend` field of a `spi_driver` structure, and thus no other messages, of any kind, that derive from use of this field, *i.e.*, that contain this field in their input set, are needed.

Finally, various kinds of problems can trigger errors that relate to the code structure. An example is the error about missing braces in line 4 of Figure 1. Our hypothesis is that the driver to port compiles correctly with its original Linux version, and thus such structural errors should be side effects of other errors. For certain kinds of messages, all other errors of certain kinds found in the same block or function are discarded.

In our example, gcc-reduce retains only the error on lines 1-3 and the one on lines 8-10 for further processing.

4.2 Patch query generation

After reducing the error messages, gcc-reduce creates a patch query for each of the remaining errors. The various kinds of errors are limited, as are the kinds of information found in their keywords. Accordingly, the patch queries can be generated by instantiating a small set of templates. 17 templates, incorporating PQL best practices, are available in our current prototype. Templates are typically parameterized by type names, *e.g.*, `spi_driver` in our example, and global function and field names, *i.e.*, terms that are common to the kernel rather than specific to the driver. gcc-reduce also generates a makefile and a document that the maintainer can use to track the changes required. A few error types are not supported by our current set of templates. In these cases, the driver maintainer can study the provided templates and produce a patch query by analogy.

5 Evaluation

The goals for our evaluation are to assess the degree to which our approach satisfies the following properties:

- Our approach is efficient enough for interactive use on a standard laptop.
- `gcc-reduce` eliminates redundant compiler error messages, but keeps the errors needed to motivate a complete forward or back port of a driver.
- The commits selected by Prequel help solve forward and back porting problems.

[4] Subsequently, we refer to compiler errors and warnings collectively as errors.

[5] http://llvm.linuxfoundation.org/index.php/Main_Page; The section "Current patch statistics" is dated 2015-01-28.

- Our approach gives more relevant results than existing approaches, such as commit history search using git or Internet search using Google.

Our evaluation focuses on drivers introduced in 2013 and 2015 and targets Linux 4.6, released on May 15, 2016, to illustrate the behavior of our approach over a longer and shorter time period. From January 1, 2013 to May 15, 2016 there were 219,879 commits to the Linux kernel and from January 1, 2015 to May 15, 2016 there were 85,812 commits to the Linux kernel. We first present our dataset and then address the above properties.

5.1 Dataset

We consider an introduced driver to be a collection of C and header files that are added into the kernel in the `drivers` directory in a single commit, accompanied by changes in a Makefile and other files related to the build infrastructure. We ignore `drivers/staging` code, as such drivers are considered to be immature and thus may contain idiosyncratic coding strategies for which example changes may not be available. We check also that the driver compiles without errors or warnings at the point where it is committed and that all of the files added or modified by the commit exist in our target version, Linux 4.6. Finally, for our forward porting experiments, for each selected commit, we overwrite the corresponding C and header files in a clean Linux 4.6 and force the compilation of each C file. We include in our data set drivers for which this compilation produces at least one warning or error. Likewise, for our back porting experiments, we take the Linux 4.6 versions of the driver files introduced in 2013 or 2015 and copy them back to the kernel version just following the commit in which the files were introduced, keeping for further analysis the files in which compilation with the older version produces errors.

Table 1 shows the number of drivers and driver `.c` files that raise porting issues. Table 2 shows the distribution of these drivers over the various driver types. The difference in the number of drivers and driver files considered in the forward and back porting cases is due to the latency of interface deprecation in the Linux kernel. An outdated and a modern interface may coexist in a newer or older kernel, in which case forward porting or back porting, respectively, is not necessary. Our approach only aims to produce a driver that is compatible with the target kernel version, and does not aim to ensure that the resulting driver uses the most recent interfaces, if the interfaces from the source kernel version remain available.

We use the complete set of drivers in our dataset for the evaluations that are fully automatic. For our porting experiments, we use only a subset, due to time constraints.

Table 1: Drivers and driver files that raise porting issues

	forward port		back port	
	drivers	driver files	drivers	driver files
2013	108	135	130	149
2015	97	119	114	125

Table 2: Distribution of considered driver files (f = forward ports, b = back ports)

	2013f	2013b	2015f	2015b		2013f	2013b	2015f	2015b		2013f	2013b	2015f	2015b		2013f	2013b	2015f	2015b
acpi	0	0	1	0	fmc	1	1	0	0	mailbox	0	0	0	2	powercap	1	0	0	0
ata	1	1	0	1	gpio	5	5	5	7	md	3	2	2	2	pwm	1	1	0	0
base	0	0	0	1	gpu	6	1	6	8	media	6	6	5	2	regulator	5	4	0	0
block	5	3	0	0	hid	0	0	1	1	mfd	2	4	0	4	reset	0	0	0	4
char	3	4	0	1	hwmon	0	3	1	1	misc	1	1	0	0	rtc	2	10	0	0
clk	6	6	10	15	hwtracing	0	0	2	3	mmc	1	2	2	2	scsi	3	2	3	1
clocksrc.	3	4	2	3	i2c	2	4	0	1	mtd	0	0	5	3	soc	0	0	2	1
cpufreq	1	1	0	0	iio	22	32	2	2	net	14	8	18	14	spi	3	1	0	0
crypto	0	0	1	2	infiniband	6	0	0	0	nfc	2	3	4	5	thermal	5	6	6	6
devfreq	0	0	1	1	input	3	5	1	1	phy	4	2	2	2	usb	1	2	1	1
dma	0	1	0	3	irqchip	0	0	1	1	pinctrl	2	1	3	6	video	1	4	0	0
edac	0	1	0	0	leds	0	1	3	2	platform	4	3	2	2	virtio	0	0	0	1
extcon	0	0	3	2	lightnvm	0	0	3	3	power	6	8	8	4	watchdog	0	2	2	5

5.2 Methodology

For a chosen driver, we apply our approach to collect relevant commits. From these commits, we manually infer the required changes and update the original driver accordingly. We consider an experiment to be a success if the changes we have made in the driver are also found in the target version of the driver and the resulting driver compiles in the target version. We do not aspire to produce code identical to the target version, because the driver may also undergo changes that are specific to its behavior, which go beyond the porting task.

We prepare indices for Prequel covering all commits that modify at least one file between Linux 3.0, released in July 2011, and Linux 4.6, amounting to 306,242 commits. The largest index is the third (Section 3.2), at 160MB. Using indices starting with 2011 is overkill when porting drivers from 2013 at the earliest. We envision, however, that developers will generate new indices only occasionally, and thus may have available a larger index than the minimal one needed for a given task. We also prepare an index of our reference version (see Section 3.2), Linux 4.6. This index is used when a keyword is needed to reduce the set of commits to which a Prequel query should be applied, but there is no keyword in a patch query that is expected to occur in or near a changed line.

5.3 Performance

Figure 5 shows the execution times of the most time-consuming parts of Prequel: commit selection and application of the patch query to the selected commits. Experiments are carried out on a single core of an Intel i5-

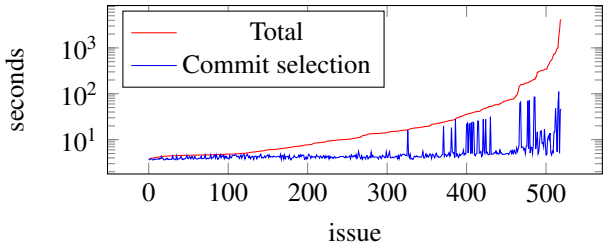


Figure 5: Execution time of Prequel on 2013 forward port issues (Total = patch query application time + commit selection time)

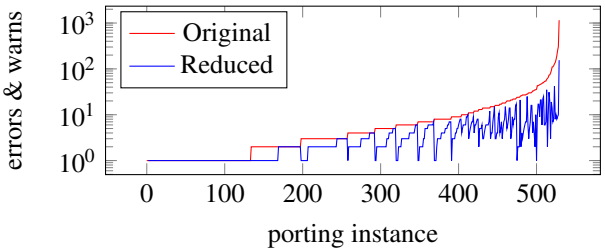


Figure 6: Reduction in the number of errors and warnings achieved by gcc-reduce

6200U 2.30GHz CPU. The machine has 12G memory. The Linux kernel and all temporary files are placed in an in-memory file system to avoid disk access costs. Each point in the graph represents a single porting issue. Issues are ordered from shortest execution time to largest. At each x-axis coordinate, the height of the blue (lower) point represents the commit selection time, and the difference between the red (upper) point and the blue point represents the patch query application time.

For 73% of the issues, the total of commit selection time and patch query application time is 30 seconds or less. Beyond that point, the commit selection time is occasionally high, up to 111 seconds, typically when the reference version is used as a last resort to reduce the number of possible commits. The average commit selection time is under 7 seconds. Patch query application time rises with the number of files in the selected commits and the file size. The maximum patch query time per commit is under 14 seconds. Further performance improvements require finding low cost ways of discarding more commits, before applying the patch query.

The overall running time of the approach for a particular driver depends on the number of issues involved. This is managed by gcc-reduce. Figure 6 shows in the red (top) line the number of errors and warnings indicated by gcc for all of the considered porting problems (2013 and 2015 drivers, forward and backward porting) and in the blue (bottom) line the number resulting from the reduction process. Each point represents a driver, and the drivers are ordered from the smallest to the largest number of gcc messages. 36% of the ports involve only one issue. 76% involve fewer than 5. For a driver with 4 issues, each taking 30 seconds or less, we thus obtain a typical patch query time of 2 minutes or less.

Finally, Table 3 shows the distribution of error types, for driver subdirectories with more than 30 errors in at least one porting experiment, as well as the average commit selection and patch query application time (Avg PQ) for each error type, as observed on the 2013 forward porting experiments (Figure 5).

Table 3: Distribution of error types

	Unknown	Unknown function	Unknown variable	Unknown type	Unknown field	Arg error	Expression type change	Field type change
char	1	22	43	4	8	4	0	0
clk	104	47	65	2	12	3	0	5
gpio	1	36	15	1	30	3	4	0
gpu	1	18	7	20	16	19	0	8
iio	13	64	31	24	35	21	3	25
infiniband	1	2	4	11	17	2	0	3
lightnvm	2	7	7	1	26	10	0	2
misc	10	16	6	1	2	4	0	0
net	6	55	49	20	55	15	1	21
platform	2	20	17	31	11	4	0	0
power	6	12	3	16	60	24	0	0
Avg PQ (sec)	N/A	25	7	9	57	197	14	267

5.4 Precision

We have used our approach in the porting of 33 driver files: 13 from their original versions in 2013 to Linux 4.6, 10 from Linux 4.6 back to their commit of introduction in 2013, and 10 from their original versions in 2015 to Linux 4.6. Gcc-reduce reduced the compiler errors and warnings associated with these drivers to 107.

For 80 of the identified issues, we were able to replicate the change as found in the Linux kernel code. For 24 issues, we encountered some kind of failure. In 6 cases, gcc-reduce misclassified an issue. For example, for two issues related to forward porting the 2013 Xen TPM frontend driver, introduced in commit e268395, gcc-reduce expects a change in the field in the case of an incompatible field initialization, while the actual issue is a change in declaration of the initial value. Gcc-reduce could create patch queries that consider more possibilities at the cost of a higher patch query time. In the same driver, there is also the only instance of a failure of Prequel: the issue is related to a macro whose uses Prequel is unable to parse, implying that no results are returned.

In 5 cases, there is more than one change to a particular code fragment between the original version of the code and the target of the porting task. For example, the changes to the Sharp GP2AP020A00F Proximity/ALS sensor driver between its original version in 2013 (commit bf29fbe) and Linux 4.6, in part involve first append-

ing _new to a set of structure fields, to allow old and new versions of the fields to co-exist, and then removing the _new in a later commit, once all relevant drivers have been updated. Prequel only finds the commit that adds the _new suffix, thus giving only a partial view of the required changes. In the remaining failure cases, the commits available are not sufficient to decide how to transform a particular piece of code. For example, back porting the Nuvoton NAU7802 ADC driver from its Linux 4.6 version to the Linux kernel as of commit 8b20be8 involves removing calls to `reinit_completion`. Many examples are available, but they involve different transformations, and indeed the commit log message indicates that many of them are bug fixes. Using a bug fix as a model for backporting would amount to introducing a bug. Thus, it is not clear from the examples which strategy is appropriate for the given driver.

Overall, we were able to address 3/4 of the issues successfully, without having any specific prior knowledge of the drivers concerned. Furthermore, doing so typically required looking at very few commits. For 86% of the successfully addressed issues, it was sufficient to look at only the first reported commit. We consulted at most 7 commits for a single issue.

We have also found back porting to be harder than forward porting. We have already noted the case where many examples are bug fixes. Furthermore, over time, the Linux kernel developers also tend to replace local, special-purpose coding strategies by generic APIs, when some operations or data are common to multiple drivers. We have often found it easier to introduce generic code from specific implementations, as required for forward porting, but harder to replace API calls by specialized local definitions or data structure representations where the design strategy may be specific to each developer.

5.5 Comparison with git

Git is at the foundation of many Linux kernel developers' development practices. We thus compare the information obtained by Prequel with the information that can be obtained using `git log -G` or `git log -S`.[6] We consider only the driver files for which we have carried out the full forward porting process (Section 5.4) and only the issues that we resolved successfully. In each case, we collect the commits that reference a keyword that is expected to be changed by the port; for example if the issue is an unknown `suspend` field, then we expect `suspend` to appear in the changed lines. We consider a range starting with the commit that we found most helpful and ending with Linux 4.6, to see how many commits git provides to the user before reaching the helpful one.

[6] `git log -G` finds patterns in changed lines, while `git log -S` additionally requires that the number of instances of those patterns changes. We choose an appropriate command given the error type.

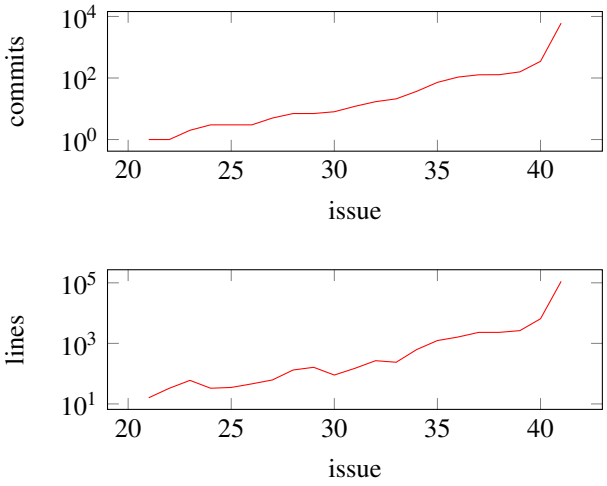


Figure 7: Commits (top) and commit lines (bottom) obtained with git log -G or git log -S

Each git command took around 90 seconds. Figure 7 shows the number of commits returned (top) and the number of lines in these commits (bottom), including both log messages and code changes. Issues are ordered by increasing number of commits, in both cases. For half of the issues (1-20 on the x-axis) there are no results, and thus these points are omitted. No results is typical of issues such as an unknown function; once the function has been removed, no more commits will mention it. Git thus succeeds in immediately returning a commit relevant to the porting problem. Other kinds of issues such as a change in the number of arguments of a function or the type of a structure field, however, do not cause the keyword to disappear, and thus it can occur in later, irrelevant commits. Furthermore, as noted in Section 2, some names, typically those of structure fields, are reused across the kernel, and thus commits are found affecting unrelated instances. For example, the rightmost point in each graph of Figure 7 derives from a search for `dev`, which is very common both as a field name for many types of structures and as the name of a local variable. Prequel on the other hand has access to type information and other relevant context information, and can select commits more precisely. In many cases, Prequel is even more efficient than git, due to the use of indexing.

Finally, 8 issues concern the incorrect type of the initial value of a structure field. In these cases, the change is typically in the definition if the initial value, which is often driver-specific, and there is no keyword whose changes git can be used to search for.

5.6 Comparison with search engines

Our approach to driver porting relies on searching through the commit history for information on how to resolve compiler errors resulting from out of date code. In many other software development contexts, developers and users turn to search engines such as Google for hints on how to address error messages. To assess the potential benefits of using an existing general-purpose search engine to address Linux driver porting issues, we have done a small experiment using Google.

Setup. It is impossible to anticipate every Google query that a developer might make. We take the straightforward solution of using the compiler error message itself (see Figure 1) as a query. In this, we drop the position information (file, etc.), which is likely too restrictive, unless someone has already ported the same driver.

Some error descriptions, however, are generic, such as the message "initialization from incompatible pointer type", found in the middle of Figure 1, that does not contain any information specific to the error context. As illustrated in Figure 1, gcc error messages either contain a code snippet, or implicitly inherit the code snippet of a previous message. Thus, we additionally consider concatenating the error description and the code snippet to form another possible query, providing more information but incurring the risk of overspecification to the targeted driver. We consider only the subset of error messages generated by our error message reduction process (Section 4.1), as the user could manually filter out the important messages, as done by our tool.

In order to assess the information provided by Google on a large scale, we use the `curl` library [3] to script requests. We then parse each resulting search result summary page to extract the entries, consisting of a title, the link as shown to the user, and the description, found on the first page of results. These tests involve our complete dataset (Section 5.1).

To assess the results, we use the measure *query recall rate at N*, meaning the percentage of cases for which the top N Google results contain at least one result that is relevant to the porting problem. A result is considered relevant if it contains all of the keywords identified by gcc-reduce for the issue. As shown in Figure 8, based on this criterion, the first result appears to be relevant only 12% of the time in the no code case, where we include only the error message. The rate of at least one relevant result rises to only 33% in the no code case if one considers the top 6 results. Even a 33% success rate is not very useful in practice, and even if the information on the summary page suggests success, there is no guarantee that the information on the linked page will actually turn out to be useful. In contrast, for the ports we have carried

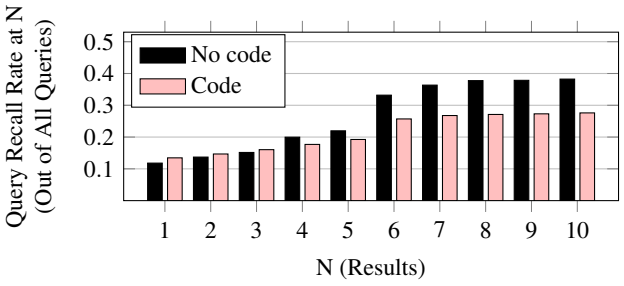


Figure 8: Rate of existence of a relevant result in the top N returns returned by Google

out, in 86% of the cases, we resolved an issue by looking only at Prequel's top ranked commit. Furthermore, there is contradictory feedback about whether it is desirable to include the code snippet; when looking at the top 1-3 results, doing so slightly increases the rate of success, but when considering a larger set of results, including the code snippet is a disadvantage. So the driver maintainer may have to launch several queries to see which is the best one. Overall, general-purpose search engines do not appear to be a promising tool for finding information about Linux kernel interface changes.

6 Limitations

Our approach is based on several hypotheses. First, we assume that the driver to port works correctly with respect to the Linux kernel version for which it has been developed. Thus, the goal of porting is to preserve its behavior. Second, we make a similar assumption about the commits we encounter; their changes preserve correctness. Third, only one change is required per issue to reach code compatible with the target version. Fourth, all relevant issues are highlighted by gcc.

We have encountered violations of the last three hypotheses in practice. A violation of the second hypothesis is illustrated by the `reinit_completion` case in Section 5.4; commits may fix bugs rather than preserve correctness. Several other failures among our 33 ported driver files arose from a violation of the third hypothesis: the complete change was broken up into several steps, and Prequel only returned commits reflecting one step in the series, thus not giving enough information to achieve compatability with the target version. A solution could be to iterate our approach. Finally, our motivating example in Section 2 violates the fourth hypothesis. In addition to the changes in the `suspend` and `resume` fields, the configuration variable `CONFIG_PM` is renamed to `CONFIG_PM_SLEEP`. Although the commit shown in Figure 2 does contain the new configuration variable, the driver maintainer has to be self-motivated to look at it;

gcc does not raise warnings about configuration variables that no longer exist. Tools, such as Undertaker [11], that check for configurability errors could be used in place of gcc within our approach to address this issue.

Our approach provides guidance on how to evolve a driver in concert with other similar drivers, but does not directly address the case where a new kernel feature could better support a unique feature of the device. Our hypothesis, however, is that new features are added to the kernel to support drivers that are already in the kernel, and that the developer adding such a specialized kernel feature will update some of the relevant drivers, if only to validate that the new feature works as expected. The commit logs and change examples identified by Prequel should then still provide guidance on how to apply a new kernel feature in another specialized situation.

7 Related work

The traditional strategy for back porting device drivers involves inserting `#ifdefs` in the driver code to implement different behaviors for different kernel versions. The Linux kernel backports project, initiated in 2007, introduced the use of a compatibility library that abstracts over the variations in different kernels. Rodriguez and Lawall [9] explored the use of Coccinelle to automate the changes needed in a driver to target this compatibility library, and this approach is now actively used by the Linux kernel backports project. These approaches require the developer to manually identify the changes needed for each kernel version, either to modify the driver code directly or to create the compatibility library.

Thung et al. [12] also target automating the backporting of Linux device drivers. Their approach identifies the commit between the source and target versions at which the driver ceases to successfully compile, and then infers transformation rules from the set of changes performed by that commit. While their approach goes further than ours, by inferring transformation rules, it is limited to the information available in the commit that breaks compilation, it has only been evaluated on pairs of successive Linux kernel releases, and it is further limited to drivers in which the compiler signals only one error line. These constraints are not satisfied by many porting issues. For example, for our motivating example in Section 2, there are multiple compiler errors, it was necessary to consult multiple commits, and the commit that breaks compilation does not contain any relevant change examples. Our approach does, however, also assume that only one change is needed for each issue to get from the source version to the target version.

Several recent approaches automate the identification of API evolutions based on analysis of changes in call-graph dependencies [7, 14]. These approaches are well-suited for porting issues that involve only the names of called functions, but not the other types of changes (field type change, etc.) that we have encountered.

Martinez et al. [6] propose a patch query language with the goal of collecting statistics on the frequency of various kinds of code changes to guide automated software repair [13]. Their approach builds on the information about occurrences of a fixed set of change types collected by ChangeDistiller [4]. Change types refer to various syntactic categories, such as removal of an `if` statement, but do not contain information about concrete terms such as function names or structure fields, as we require to limit the results to the commits relevant to a given porting problem. The approach furthermore focuses solely on changes, and thus does not allow queries on context code, as we have also found essential. Stevens et al. [10] propose a query language for changes identified by ChangeDistiller, relying on a logic-programming-based notation. They use their approach for studying instances of refactorings, rather than porting problems.

8 Conclusion

Porting device drivers is an ever-present problem in the context of the Linux kernel. A major challenge in the porting process is to obtain adequate information as to how the port should be carried out. Indeed, the Linux kernel interface is huge, and many relevant details about an interface change are only known to the specific maintainer who has carried it out.

In this paper we have proposed an approach to extract information from compiler output and a git commit history about where changes are needed and how to carry those changes out. On 33 driver files, exhibiting 107 porting issues, our approach enabled us to address 3/4 of the issues, with no specific knowledge about the drivers concerned. Our approach is also reasonably efficient, producing complete results for a driver in at most a few minutes for many cases.

Future work will involve improving performance and addressing the identified limitations, such as the requirement of only one change per issue between the source and target versions. Inferring changes automatically from examples would then be the next major step.

Acknowledgments. We thank the anonymous reviewers and our shepherd Daniel Williams for their feedback on the paper. This work is supported in part by OSADL and by ANR ITrans.

Availability. Our tools and the 33 driver file experiments are available at http://prequel-pql.gforge.inria.fr/

References

[1] APPEL, A. W. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[2] 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* (2009), pp. 114–126.

[3] Curl. https://curl.haxx.se/.

[4] FLURI, B., AND GALL, H. C. Classifying change types for qualifying change couplings. In *14th IEEE International Conference on Program Comprehension* (2006), pp. 35–45.

[5] LAWALL, J., LAMBERT, Q., AND MULLER, G. Prequel: A patch-like query language for commit history search. Research Report RR-8918, Inria Paris, June 2016.

[6] MARTINEZ, M., DUCHIEN, L., AND MONPERRUS, M. Automatically extracting instances of code change patterns with AST analysis. In *ICSM* (2013), pp. 388–391.

[7] MENG, S., WANG, X., ZHANG, L., AND MEI, H. A history-based matching approach to identification of framework evolution. In *ICSE* (2012), pp. 353–363.

[8] 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.

[9] RODRIGUEZ, L. R., AND LAWALL, J. Increasing automation in the backporting of Linux drivers using Coccinelle. In *11th European Dependable Computing Conference - Dependability in Practice (EDCC)* (2015), pp. 132–143.

[10] STEVENS, R., AND ROOVER, C. D. Extracting executable transformations from distilled code changes. In *Software Analysis, Evolution, and Reengineering (SANER)* (2017), pp. 171–181.

[11] TARTLER, R., LOHMANN, D., SINCERO, J., AND SCHRÖDER-PREIKSCHAT, W. Feature consistency in compile-time-configurable system software: facing the Linux 10,000 feature problem. In *EuroSys* (2011), pp. 47–60.

[12] THUNG, F., BACH, L. D. X., LO, D., AND LAWALL, J. Recommending code changes for automatic backporting of Linux device drivers. In *ICSME* (2016), pp. 222–232.

[13] WEIMER, W., NGUYEN, T., LE GOUES, C., AND FORREST, S. Automatically finding patches using genetic programming. In *ICSE* (2009), pp. 364–374.

[14] WU, W., GUÉHÉNEUC, Y.-G., ANTONIOL, G., AND KIM, M. AURA: a hybrid approach to identify framework evolution. In *ICSE-Volume 1* (2010), pp. 325–334.

Optimizing the TLB Shootdown Algorithm with Page Access Tracking

Nadav Amit
VMware Research

Abstract

The operating system is tasked with maintaining the coherency of per-core TLBs, necessitating costly synchronization operations, notably to invalidate stale mappings. As core-counts increase, the overhead of TLB synchronization likewise increases and hinders scalability, whereas existing software optimizations that attempt to alleviate the problem (like batching) are lacking.

We address this problem by revising the TLB synchronization subsystem. We introduce several techniques that detect cases whereby soon-to-be invalidated mappings are cached by only one TLB or not cached at all, allowing us to entirely avoid the cost of synchronization. In contrast to existing optimizations, our approach leverages hardware page access tracking. We implement our techniques in Linux and find that they reduce the number of TLB invalidations by up to 98% on average and thus improve performance by up to 78%. Evaluations show that while our techniques may introduce overheads of up to 9% when memory mappings are never removed, these overheads can be avoided by simple hardware enhancements.

1. Introduction

Translation lookaside buffers (TLBs) are perhaps the most frequently accessed caches whose coherency is not maintained by modern CPUs. The TLB is tasked with caching virtual-to-physical translations ("mappings") of memory addresses, and so it is accessed upon every memory read or write operation. Maintaining TLB coherency in hardware hampers performance [33], so CPU vendors require OSes to maintain coherency in software. But it is difficult for OSes to *efficiently* achieve this goal [27, 38, 39, 41, 48].

To maintain TLB coherency, OSes employ the TLB shootdown protocol [8]. If a mapping m that possibly resides in the TLB becomes stale (due to memory mapping changes) the OS flushes m from the local TLB to restore coherency. Concurrently, the OS directs remote cores that might house m in their TLB to do the same, by sending them an inter-processor interrupt (IPI). The remote cores flush their TLBs according to the information supplied by the initiator core, and they report back when they are done. TLB shootdown can take microseconds, causing a notable slowdown [48]. Performing TLB shootdown in hardware, as certain CPUs do, is faster but still incurs considerable overheads [22].

In addition to reducing performance, shootdown overheads can negatively affect the way applications are constructed. Notably, to avoid shootdown latency, programmers are advised against using memory mappings, against unmapping them, and even against building multithreaded applications [28, 42]. But memory mappings *are* the efficient way to use persistent memory [18, 47], and avoiding unmappings might cause corruption of persistent data [12].

OSes try to cope with shootdown overheads by batching them [21, 43], avoiding them on idle cores, or, when possible, performing them faster [5]. But the potential of these existing solutions is inherently limited to certain specific scenarios. To have a generally applicable, efficient solution, OSes need do know which mappings are cached by which cores. Such information can in principle be obtained by replicating the translation data structures for each core [11], but this approach might result in significantly degraded performance and wasted memory.

We propose to avoid unwarranted TLB shootdowns in a different manner: by monitoring access bits. While TLB coherency is not maintained by the CPU, CPU architectures can maintain the consistency of access bits, which are set when a mapping is cached. We contend that these bits can therefore be used to reveal which mappings are cached by which cores. To our knowledge, we are the first to use access bits in this way.

In the x86 architecture, which we study in this paper, access bit consistency is maintained by the memory subsystem. Exploiting it, we propose techniques to identify two types of common mappings whose shootdown can be avoided: (1) short-lived private mappings, which are only cached by a single core; and (2) long-lived idle mappings, which are reclaimed after the corresponding pages have not been used for a while and are not cached at all. Using

these techniques, we implement a fully functional prototype in Linux 4.5. Our evaluation shows that our proposal can eliminate more than 90% of TLB shootdowns and improve the performance of memory migration by 78%, of copy-on-write events by 18–25%, and of multithreaded applications (Apache and parallel bzip2) by up to 12%.

Our system introduces a worst case slowdown of up to 9% when mappings are only set and never removed or changed, which means no shootdown activity is conducted. This slowdown is caused, according to our measurements, due to the overhead of our TLB manipulation software techniques. To eliminate it, we propose a CPU extension that would allow OSes to write entries directly into the TLB, and resembles the functionality provided by CPUs that employ software-TLB.

2. Background and Motivation

2.1 Memory Management Hardware

Virtual memory is supported by most modern CPUs and used by all the major OSes [9, 32]. Using virtual memory allows the OS to utilize the physical memory more efficiently and to isolate the address space of each process. The CPU translates the virtual addresses to physical addresses before memory accesses are performed. The OS sets the virtual address translations (also called "mappings") according to its policies and considerations.

The memory mappings of each address space are kept in a memory-resident data structure, which is defined by the CPU architecture. The most common data structure, used by the x86 architecture, is a radix-tree, which is also known as a page-table hierarchy. The leaves of the tree, called the page-table entries (PTEs), hold the translations of fixed-sized virtual memory pages to physical frames. To translate a virtual address into a physical address, the CPU incorporates a memory management unit (MMU), which performs a "page table walk" on the page table hierarchy, checking access permissions at every level. During a page-walk, the MMU updates the status bits in each PTE, indicating whether the page was read from and/or written to (dirtied).

To avoid frequent page-table walks and their associated latency, the MMU caches translations of recently used pages in a translation lookaside buffer (TLB). In the x86 architecture, these caches are maintained by the hardware, bringing translations into the cache after page walks and evicting them according to an implementation-specific cache replacement policy. Each x86 core holds a logically private TLB.

Unlike memory caches, TLBs of different CPUs are not maintained coherent by hardware. Specifically, x86 CPUs do not maintain coherence between the TLB and the page-tables, nor among the TLBs of different cores. As a result, page-table changes may leave stale entries in the TLBs until coherence is restored by the OS. The instruction set enables the OS to do so by flushing ("invalidating") individual PTEs or the entire TLB. Global and individual TLB flushes can only be performed locally, on the TLB of the core that executes the flush instruction.

Although the TLB is essential to attain reasonable translation latency, some workloads experience frequent TLB cache-misses [4]. Recently, new features were introduced into the x86 architecture to reduce the number and latency of TLB misses. A new instruction set extension allows each page-table hierarchy to be associated with an address-space ID (ASID) and avoid TLB flushes during address-space switching, thus reducing the number of TLB misses. Micro-architectural enhancements introduced page-walk caches that enable the hardware to cache internal nodes in the page-table hierarchy, thereby reducing TLB-miss latencies [3].

2.2 TLB Software Challenges

The x86 architecture leaves maintaining TLB coherency to the OSes, which often requires frequent TLB invalidations after PTE changes. OS kernels can make such PTE changes independently of the running processes, upon memory migration across NUMA nodes [2], memory deduplications [49], memory reclamation, and memory compaction for accommodating huge pages [14]. Processes can also trigger PTE changes by using system calls, for example `mprotect`, which changes protection on a memory range, or by writing to copy-on-write pages (COW).

These PTE changes can require a TLB flush to avoid caching of stale PTEs in the TLB. We distinguish between two types of flushes: *local* and *remote*, in accordance with the core that initiated the PTE change. Remote TLB flushes are significantly more expensive, since most CPUs cannot flush remote TLBs directly. OSes therefore perform a *TLB shootdown*: The initiating core sends an inter-processor interrupt (IPI) to the remote cores and waits for their interrupt handlers to invalidate their TLBs and acknowledge that they are done.

TLB shootdowns introduce a variety of overheads. IPI delivery can take several hundreds of cycles [5]. Then, the IPI may be kept pending if the remote core has interrupts disabled, for instance while running a device driver [13]. The x86 architecture does not allow OSes to flush multiple PTEs efficiently, requiring the OS to either incur the overhead of multiple flushes or flush the entire TLB and increase the TLB miss rate. In addition, TLB flushes may

indirectly cause lock contention since they are often performed while the OS holds a lock [11, 15]. It is noteworthy that while some CPU architectures (e.g., ARM) enable to perform remote TLB shootdowns without IPIs, remote shootdowns still incur higher performance overhead than local ones [22].

2.3 OS Solutions and Shortcomings

To reduce TLB related overheads, OSes employ several techniques to avoid unnecessary shootdowns, reduce their time, and avoid TLB misses.

A TLB shootdown can be avoided if the OS can ensure that the modified PTE is either not cached in remote TLBs or can be flushed at a later time, but before it can be used for an address translation. In practice, OSes can only avoid remote shootdowns in certain cases. In Linux, for example, each userspace PTE is only set in a single address space page-table hierarchy, allowing the OS to track which address space is active on each core and flush only the TLBs of cores that currently use this address space. The TLB can be flushed during context switch, before any stale entry would be used.

A common method to reduce shootdown time is to batch TLB invalidations if they can be deferred [21, 47]. Batching, however, cannot be used in many cases, for example when a multithreaded application changes the access permissions of a single page. Another way to reduce shootdown overhead is to acknowledge its IPI immediately, even before invalidation is performed [5, 43].

Flush time can be reduced by lowering the number of TLB flushes. Flushing multiple individual PTEs is expensive, and therefore OSes can prefer to flush the entire TLB if the number of PTEs exceeds a certain threshold. This is a delicate trade-off, as such a flush increases the number of TLB misses [23].

Linux tries to balance between the overheads of TLB flushes and TLB misses when a core becomes idle, using a lazy TLB invalidation scheme. Since the process that ran before the core became idle may be scheduled to run again, the OS does not switch its address space, in order to avoid potential future TLB misses. However, when the first TLB shootdown is delivered to the idle core, the OS performs a full TLB invalidation and indicates to the other cores not to send it further shootdown IPIs while it is idle.

Despite all of these techniques, shootdowns can induce high overheads in real systems. Arguably, this overhead is one of the reasons people refrain from using multithreading, in which mapping changes need to propagate to all threads. Moreover, application writers often prefer copying data over memory remapping, which requires TLB shootdown [42].

2.4 Per-Core Page Tables

Currently, the state-of-the-art software solution for TLB shootdowns is setting per-core page tables, and according to the experienced page-faults track which cores used each PTE [11,19]. When a PTE invalidation is needed, a shootdown is sent only to cores whose page tables hold the invalidated PTE.

Maintaining per-core page tables, however, can introduce substantial overheads when some PTEs are accessed by multiple cores. In such a case, OS memory management operations become more expensive, as mapping modifications require changes the of PTEs in multiple page-tables. The overhead of PTE changes is not negligible, as some require atomic operations. RadixVM [11] reduces this overhead by changing PTEs in parallel: sending IPIs to cores that hold the PTE and changing them locally. This scheme is efficient when shootdowns are needed, as one IPI triggers both the PTE change and its invalidation. Yet, if a shootdown is not needed, for example when the other cores run a different process, this solution may increase the overhead due to the additional IPIs.

Holding per-core page tables can also introduce high memory overheads if memory is accessed by multiple cores. For example, in recent 288 core CPUs [24], if half of the memory is accessed by all cores, the page tables will consume 18% of the memory or more if memory is overcommitted or mappings are sparse.

While studies showed substantial performance gains when per-core page tables are used, the limitations of this approach may have not been studied well enough. For example, in an experiment we conducted memory migration between NUMA nodes was 5 times slower when memory was mapped in 48 page-table hierarchies (of 48 Linux running processes in our experiment) instead of one. Previous studies may have not shown these overheads as they considered a teaching OS, which lacks basic memory management features [11]. In addition, previous studies experienced shootdown latencies of over 500k cycles, which is over 24x of the latency that we measured. Presumably, the high overhead of shootdowns could overshadow other overheads.

3. The Idea

The challenge in reducing TLB shootdown overhead is determining which cores, if at all, might be caching a given PTE. Although architectural paging structures do not generally provide this information, we contend that the OS can nevertheless deduce it by carefully tracking and manipulating PTE access-bits. The proclaimed goal of access bits is to indicate if memory pages have been accessed. This functional-

ity is declared by architectural manuals and is used by OSes to make informed swapping decisions. Our insight is that access bits can be additionally used for a different purpose: to indicate if PTEs are cached in TLBs, as explained next.

Let us assume: that (1) a PTE e might be cached by a set of cores S at time t_0; that (2) e's access bit is clear at t_0 (because it was never set, or because the OS explicitly cleared it); and that (3) this bit is still clear at some later time t_1. Since access bits are set by hardware whenever it caches the corresponding translations in the TLB [25], we can safely conclude that e is *not* cached by any core $c \notin S$ at t_1.

We note that our reasoning rests on the fact that last-level TLBs are private per core [6, 27, 29] and so translations are not transferred between them. Linux, for example, relies on this fact when shooting down a PTE of some address space α while avoiding the shootdown at remote cores whose current address spaces are different than α (§2.3). This optimization would have been erroneous if TLBs were shared, because Linux permits the said remote cores to freely load α while the shootdown takes place, which would have allowed them to cache stale mappings from a shared last-level TLB, thereby creating an inconsistency bug.

We identify two types of mappings that can help us optimize TLB shootdown by leveraging access-bit information. The first is *short-lived private mappings* of pages that are accessed exclusively by a single thread and then removed shortly after; this access pattern may be exhibited, for example, by multithreaded applications that use memory-mapped files to read data. The second type is *long-lived idle mappings* of pages that are reclaimed by the OS after they have not been accessed for a while; this pattern is typical for pages that cease to be part of the working set of a process, prompting the OS to unmap them, flush their PTEs, and reuse their frames elsewhere.

4. The System

Using the above reasoning (§3), we next describe the Linux enhancements we deploy on an x86 Intel machine to optimize TLB shootdown of short-lived private mappings (§4.1) and long-lived idle mappings (§4.2). We then describe "software-PTEs", the data structures we use when implementing our mechanisms (§4.3). To distinguish our enhancements from the baseline OS, we collectively denote them as ABIS—access-based invalidation system.

4.1 Private PTE Detection

To avoid TLB shootdown due to a private mapping, we must (1) identify the core that initially uses this mapping and (2) make sure that other cores have not used it too at a later time. As previously shown [27], the first item is achievable via demand paging, the standard memory management technique that OSes employ, which traps upon the first access to a memory page and only then sets a valid mapping [9]. The second item, however, is more challenging, as existing approaches to detect PTE sharing can introduce overheads that are much higher than those we set out to eliminate (§6).

Direct TLB Insertion Our goal is therefore to find a low-overhead way to detect PTE sharing. As a first step, we note that this goal would have been easily achievable if it was possible to conduct *direct TLB insertion*—inserting a mapping m directly into a TLB of a core c without setting the access bit of the corresponding PTE e. Given such a capability, as long as m resides in the TLB, subsequent uses of m by c would not set the access-bit of e, as no page table walks are needed. In contrast, if some other core $\bar{c}$ ends up using m as well, the hardware will walk the page table when inserting m to the TLB of $\bar{c}$, and it will therefore set e's access bit, thereby indicating that m is not private.

Direct TLB insertion would have thus allowed us to use turned-off access bits as identifiers of private mappings. We remark that this method is best-effort and might lead to false-positive indications of sharing in cases where m is evicted from the TLB and reinserted later. This issue does not affect correctness, however. It simply implies that some useless shootdown activity is possible. The approach is thus more suitable for short-lived PTEs.

Alas, current x86 processors do not support direct TLB insertion. One objective of this study is to motivate such support. When proposing a new hardware feature, architects typically resort to simulation since it is unrealistic to fabricate chips to test research features. We do not employ simulation for two reasons. First, because we suspect that it might yield questionable results, as the OS memory management subsystems that are involved are complex to realistically simulate. Second, since TLB insertion is possible on existing hardware even without hardware support, and can benefit workloads that are sensitive to shootdown overheads, shortening runtimes by 0.56x ($=\frac{1}{1.78}$; see Figure 5) at best. Although runtimes might be 1.09x longer in the worst case, our results indicate that real hardware support will eliminate this overhead (§5.1).

Note that although direct TLB insertion is not supported in the x86 architecture, it is supported in CPUs that employ software-managed TLBs. For example, Power CPUs support the `tlbwe` instruction

that can insert PTE directly into the TLB. We therefore consider this enhancement achievable with a reasonable effort.

Approximation Let us first rule out the naive approach to approximate direct TLB insertion by: (1) setting a PTE e; (2) accessing the page and thus prompting hardware to load the corresponding mapping m into the TLB and to set e's access bit; and then (3) having the OS clear e's access bit. This approach is buggy due to the time window between the second and third items, which allows other cores to cache m in their TLBs before the bit is cleared, resulting in a false sharing indications that the page is private. Shootdown will then be erroneously skipped.

We resolve this problem and avoid the above race by using Intel's address space IDs, which is known as process-context identifiers (PCIDs) [25]. PCIDs enable TLBs to hold mappings of multiple address spaces by associating every cached PTE with a PCID of its address space. The PCID of the current address space is stored in the same register as the pointer to the root of the page table hierarchy (CR3), and TLB entries are associated with this PCID when they are cached. The CPU uses for address translation only PTEs whose PCID matches the current one. This feature is intended to allow OSes to avoid global TLB invalidations during context switch and reduce the number of TLB misses.

PCID is not currently used by Linux due to the limited number of supported address spaces and questionable performance gains from TLB miss reduction. We indeed exploit this feature in a different manner. Nevertheless, our use does not prevent or limit future PCID support in the OS.

The technique ABIS employs to provide direct TLB insertion is depicted in Figure 1. Upon initialization, ABIS preallocates for each core a "secondary" page-table hierarchy, which consists of four pages, one for each level of the hierarchy. The uppermost level of the page-table (PGD) is then set to point to the kernel mappings (like all other address spaces). The other three pages are not connected at this stage to the hierarchy, but wired dynamically later according to the address of the PTE that is inserted to the TLB.

While executing, the currently running thread T occasionally experiences page faults, notably due to demand paging. When a page fault fires, the OS handler is invoked and locks the PT that holds the faulting PTE—no other core will simultaneously handle the same fault.

At this point, ABIS loads the secondary space to CR3 along with a PCID equal to that of T (Step 1 in Figure 1). After, ABIS wires the virtual-to-physical mapping of the target page in both primary and

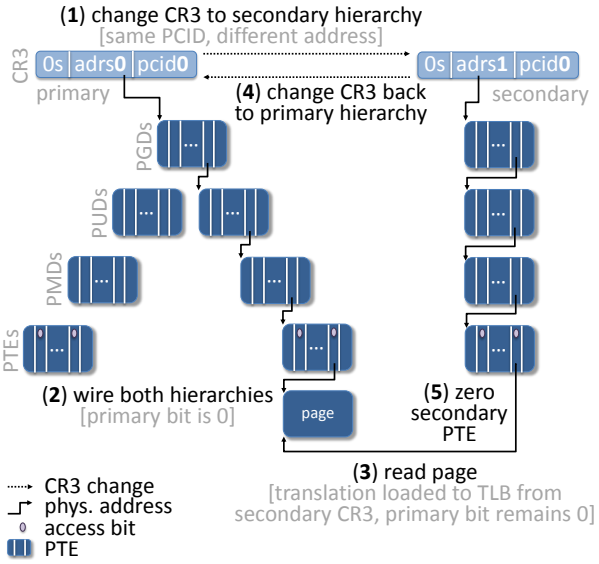


Figure 1: Direct TLB insertion using a secondary hierarchy.

secondary spaces, leaving the corresponding access bit in the primary hierarchy clear (Step 2).

Then, ABIS reads from the page. Because the associated mapping is currently missing from the TLB (a page fault fired), and because CR3 currently points to the secondary space, reading the page prompts the hardware to walk the secondary hierarchy and to insert the appropriate translation to the TLB, leaving the primary bit clear (Step 3). Importantly, the inserted translation is valid and usable within the primary space, because both spaces have the same PCID and point to the same physical page using the same virtual address. This approach eliminates he aforementioned race: no other core is able to access the secondary space, as it is private to the core.

After reading the page, ABIS loads the primary hierarchy back to CR3, to allow the thread to continue as usual (Step 4). It then clears the PTE from the secondary space, thereby preventing further use of translation data from the secondary hierarchy that may have been cached in the hardware page-walk cache (PWC). If the secondary tables are used by the CPU for translation, no valid PTE will be found and the CPU will restart a page-walk from the root entry.

Finally, using our "software-PTE" (SPTE) data structure (§4.3), ABIS associates the faulting PTE e with the current core c that has just resolved e. When the time comes to flush e, if ABIS determines that e is still private to c, it will invalidate e on c only, thus avoiding the shootdown overhead.

Coexisting with Linux Linux reads and clears architectural access bits (hwA-s) via a small API, allowing us to easily mask these bits while making sure

that both Linux and ABIS simultaneously operate correctly. Notably, when Linux attempts to clear an hwA, ABIS (1) checks whether the bit is turned on, in which case it (2) clears the bit and (3) records in the SPTE the fact that the associated PTE is not private (using the value `ALL_CPUS` discussed further below). Note, however, that Linux and ABIS can use the access bit in a conflicting manner. For example, after a page fault, Linux could expect to see the access bit turned on, whereas ABIS's direct TLB insertion makes sure that the opposite happens. To avoid any such conflicts, we maintain in the SPTE a new per-PTE "software access bit" (swA) for Linux, which reflects Linux's expectations. The swA bits are governed by the following rules: upon a page fault, we set the swA; when Linux clears the bit, we clear the swA; and when Linux queries the bit, we return an OR'd value of swA and hwA. These rules ensure that Linux *always* observes the values it would have observed in an ABIS-less system.

ABIS attempts to reduce false indications of PTE sharing when possible. We find that Linux performs excessive full flushes to reduce the number of IPIs sent to idle cores as part of the shootdown procedure (§2.3). In Linux, this behavior is beneficial as it reduces the number of TLB shootdowns at the cost of more TLB misses, whose impact is relatively small. In our system, however, this behavior can result in *more* shootdowns, as it increases the number of false indications. ABIS therefore relaxes this behavior, allowing idle cores to service a few individual PTE flushes before resorting to a full TLB flush.

Overhead Overall, the overhead of direct TLB insertions in our system is ≈550 cycles per PTE (responsible for the worst-case 9% slowdown mentioned earlier). This overhead is amortized when multiple PTEs are mapped together, for example, via one mmap system-call invocation, or when Linux serves a page-fault on a file-backed page and maps adjacent PTEs to avoid future page-faults [36].

4.2 TLB Version Tracking

Based on our observations from §3, we build a TLB version tracking mechanism to avoid flushes of long-lived idle mappings. Let us assume that a PTE e might be cached by a set of cores S at time t_0, and that each core $c \in S$ performed a full TLB flush during the time period (t_0, t_1). If at time t_1 the access bit of e remains clear (i.e., was not cleared by software), then we know for a fact e is not cached by any TLB. If the OS obtained the latter information by atomically reading and zeroing e, then all TLB flushes associated with e (local and remote) can be avoided. To detect such cases, we first need to maintain a "full-flush version number" for S, such that the version is incremented whenever *all* cores $c \in S$ perform a full TLB flush. Recording this version for each e at the time e is updated would then allow us to employ the optimization.

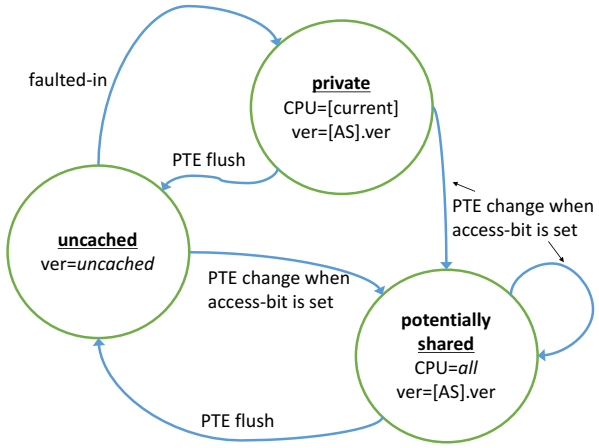


Figure 2: A finite state machine that describes the various states of a PTE. In each state, the assignment of the caching core and version are denoted. On each transition the access-bit is cleared.

TLB version tracking The most accurate way to track full flushes is by maintaining a version for each core, advancing it after each local full flush, and storing a vector of the versions for every PTE. Then, if a certain core's version differs from the corresponding vector coordinate (and the access-bit is clear), a flush on that core is not required. Despite its accuracy, this scheme is impractical, as it consumes excessive memory and requires multiple memory accesses to update version vectors. We therefore trade off accuracy in order to reduce the memory consumption of versions and the overheads of updating them.

ABIS therefore tracks versions for each address space (AS, corresponds to the above S) and not for each core. To this end, for every AS, we save a version number and a bitmask that marks which cores have not performed a full TLB flush in the current version. The last core to perform a full TLB flush in a certain version advances the version. At the same time, it marks in the bitmask which cores currently use this AS and can therefore cache PTEs in the next version. To mitigate cache line bouncing, the core that initiates a TLB shootdown updates the version on behalf the target cores.

Avoiding flushes After a PTE access-bit is cleared, ABIS stores the current AS version as the PTE version. Determining later whether a shootdown is needed requires some attention, as even if the PTE and the AS versions differ, a flush may be necessary. Consider

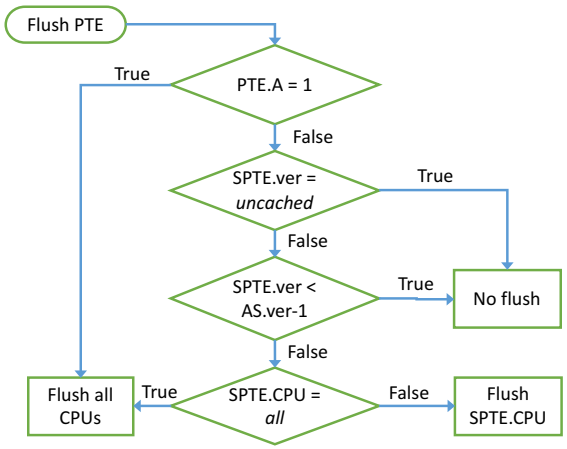


Figure 3: Flush type decision algorithm.

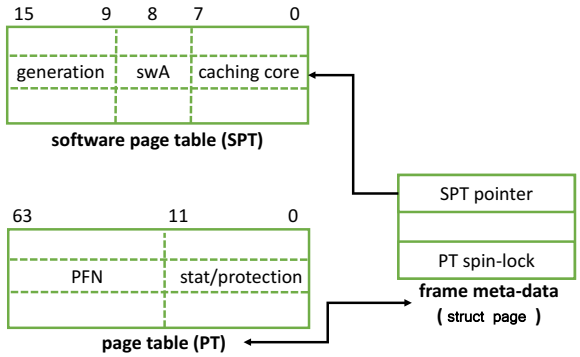


Figure 4: Software PTE (SPTE) and its association to the page table through the meta-data of the page-table frame.

a situation in which the access-bit is cleared, and the PTE version is updated to hold the AS version. At this time, some of the cores may have already flushed their TLB for the current AS version, and their respective bit in the bitmask is clear. The AS version may therefore advance before these cores flush their TLB again, and these cores can hold stale PTEs even when the versions differ. Thus, our system avoids shootdown only if there is a gap of at least one version between the AS and the PTE versions, which indicates a flush was performed on all cores.

Since flushes cannot be avoided when the access-bit is set, this bit should be cleared and the PTE version updated as frequently as possible, assuming it introduces negligible overheads. In practice, ABIS clears the bit and updates the version whenever the OS already accesses a PTE for other purposes, for example during an `mprotect` system-call or when the OS considers a page for reclamation.

Uncached PTEs The version tracking mechanism can also prevent unwarranted multiple flushes of the same PTE. Such flushes may occur, for example, when a user first calls an `msync` system call, which performs writeback of a memory mapped file, and then unmaps the file. Both operations require flushing the TLB since the first clears PTEs' dirty-bit and the second sets a non-present PTE. However, if the PTE was not accessed after the first flush, the second flush is unnecessary, regardless of whether a full TLB flush happened in between. To avoid this scenario, we set a special version value, `UNCACHED`, as the PTE version when it is flushed. This value indicates the PTE is not cached in any TLB if the access-bit is cleared, regardless of the current AS version.

Coexisting with Private PTE Detection Version tracking coexists with private PTE detection. The interaction between the two can be described in a state machine, as shown in Figure 2. In the "uncached" state a TLB flush is unnecessary; in the "private" state at most one CPU needs to perform a TLB flush; and in the "potentially shared" state all the CPUs perform TLB flush.[1] In the latter two states, a TLB flush may still be avoided if the access-bit is clear and the current address space version is at least two versions ahead of the PTE version. Figure 3 shows ABIS flush decision algorithm.

4.3 Software PTEs

As we noted before, for our system to perform informed TLB invalidation decisions, additional information must be saved for each PTE: the PTE version, the CPU which caches the PTE, and a software access-bit. Although we are capable of squeezing this information into two bytes, the architectural PTE only accommodates three bits for software use. We therefore allocate a separate "software page-table" (SPT) for each PT, which holds the corresponding "software-PTEs" (SPTEs). The SPTE is not used by the CPU during page-walks and therefore causes little cache pollution and overhead.

An SPTE is depicted in Figure 4. We use 7 bits for the version, 1 bit for the software access-bit, and another byte to track the core that caches the PTE if the access-bit is cleared. We want to define the SPTE in a manner that ensures a zeroed SPTE would behave in the legacy manner, allowing us to make fewer code changes. To do so, we reserve the zero value of the "caching core" field to indicate that the PTE may be cached by all CPUs (`ALL_CPUS`) and instead store the core number plus one.

When the OS wishes to access the SPTE of a certain PTE, it should be able to easily access it. Yet the PTE cannot accommodate a pointer to its SPTE. A possible solution is to allocate two page-frames for each page-

[1] A TLB flush is not required on CPUs that currently use a different page-table hierarchy as explained in §2

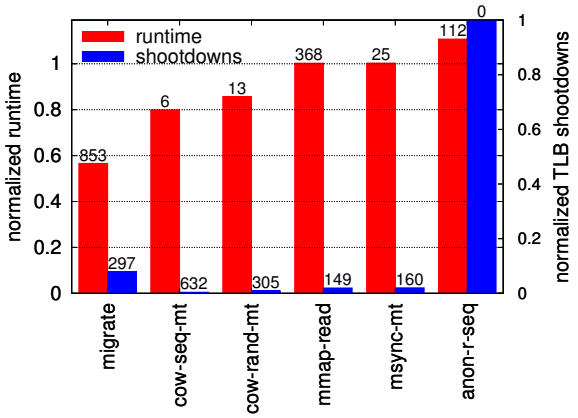


Figure 5: Normalized runtime and number of TLB shootdowns in ABIS when running vm-scalability benchmarks. The numbers above the bars indicate the baseline (left) runtime in seconds and (right) rate of TLB shootdowns in thousands/second.

table, one holding the CPU architectural PTEs and the second holding the corresponding SPTEs, each in a fixed offset from its PTE. While this scheme is simple, it wastes memory as it requires the SPTE to be the same size as a PTE (8B), when in fact SPTE only occupies two bytes.

We therefore allocate an SPT separately during the PT construction, and set a pointer to the SPT in the PT page-frame meta-data (`page struct`). Linux can quickly retrieve this meta-data, allowing us to access the SPTE of a certain PTE with small overhead. The SPTE pointer does not increase the page-frame meta-data, as it is set in an unused PT meta-data field (second quadword). The SPT therefore increases page table memory consumption by 25%. ABIS prevents races during SPT changes by protecting it with the same lock that is used to protect PT changes. It is noteworthy that although SPT management introduces a overhead, it is negligible relatively to other overheads in the workloads we evaluated.

5. Evaluation

We implemented a fully-functional prototype of the system, ABIS, which is based on Linux 4.5. As a baseline system for comparison we use the same version of Linux, which includes recent TLB shootdown optimizations. We run each test 5 times and report the average result. Our testbed consists of a two-socket Dell PowerEdge R630 with Intel 24-cores Haswell EP CPUs. We enable x2APIC cluster-mode, which speeds up IPI delivery.

In our system we disable transparent huge pages (THP), which may cause frequent full TLB flushes, increase the TLB miss-rate [4] and introduce additional overheads [26]. In practice, when THP is enabled, ABIS still shows benefit when small pages are used (e.g., in the Apache benchmark shown later) and no impact when huge pages are used (e.g., PBZIP2).

As a fast block device for our experiments we use ZRAM, a compressed RAM block device, which is used by Google Chrome OS and Ubuntu. This device latency is similar to that of emerging non-volatile memory modules. In our test, we disable memory deduplication and deep sleep states which may increase the variance of the results.

5.1 VM-Scalability

We use the vm-scalability test suite [34], which is used by Linux kernel developers to exercise the kernel memory management mechanisms, test their correctness and measure their performance.

We measure ABIS performance by running benchmarks that experience high number of TLB shootdowns.[2] To run the benchmarks in a reasonable time, we limit the amount of memory each test consumes to 32GB. Figure 5 presents the measured speedup, the runtime, the relative number of sent TLB shootdowns and their rate. We now discuss these results.

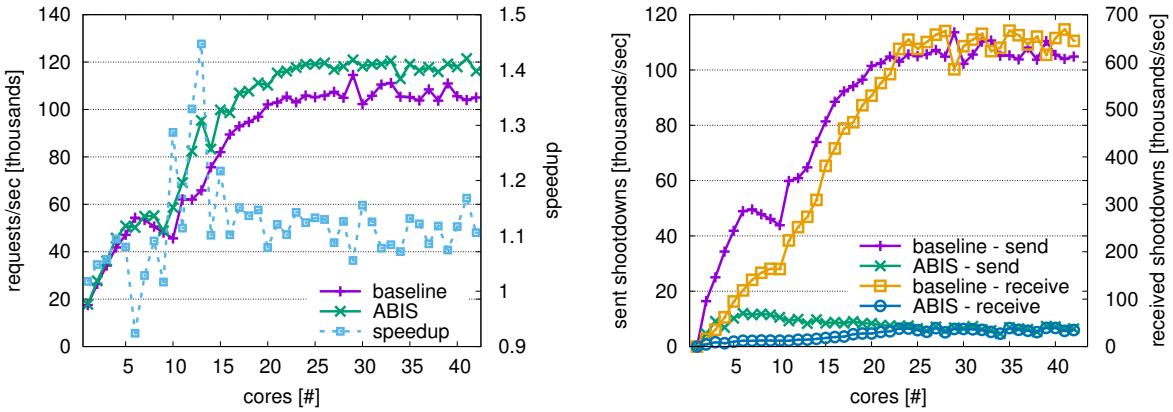
Migrate. This benchmark reads a memory mapped file and waits while the OS is instructed to migrate the process memory between NUMA nodes. During migration, we set the benchmark to perform a busy-wait loop to practice TLB flushes. We present the time that a 1TB memory migration would take. ABIS reduces runtime by 44% and shootdowns by 92%.

Multithreaded copy-on-write (cow-mt). Multiple threads read and write a private memory mapped file. Each write causes the kernel to copy the original page, update the PTE to point to the copy, and flush the TLB. ABIS prevents over 97% of the shootdowns, reducing runtime by 20% for sequential memory accesses and 15% for random by avoiding over 97%.

Memory mapped reads (mmap-read). Multiple processes read a big sparse memory mapped file. As a result, memory pressure builds up, and memory is reclaimed. While almost all the shootdowns are eliminated, the runtime is not affected, as apparently there are more significant overheads, specifically those of the page frame reclamation algorithm.

Multithreaded msync (msync-mt). Multiple threads access a memory mapped file and call the `msync` system-call to flush the memory changes to the file. `msync` can cause an overwhelming number of flushes, as the OS clears the dirty-bit. ABIS eliminates 98% of the shootdowns but does not reduce the runtime, as file system overhead appears to be the main performance bottleneck.

[2] We find that due to some benchmarks practice unrealistic scenarios. Our revised tests are released with ABIS code.



(a) Throughput (b) TLB shootdowns

Figure 6: Execution of an Apache web server which serves the Wrk workload generator.

Anonymous memory read (anon-r-seq). To evaluate ABIS overheads we run a benchmark that performs sequential anonymous memory reads and does not cause TLB shootdowns. This benchmark's runtime is 9% longer using ABIS. Profiling the benchmark shows that the software TLB manipulations consume 9% of the runtime, suggesting that hardware enhancements to manipulate the TLB can eliminate most of the overheads.

5.2 Apache Web Server

Apache is the most widely used web server software. In our tests, we use Apache v2.4.18 and enable buffered server logging for more efficient disk accesses. We use the multithreaded Wrk workload generator to create web requests [50], and set it to repeatedly request the default Apache web page for 30 seconds, using 400 connections and 6 threads. We use the same server for both the generator and Apache, and isolate each one on a set of cores. We ensure that the generator is unaffected by ABIS.

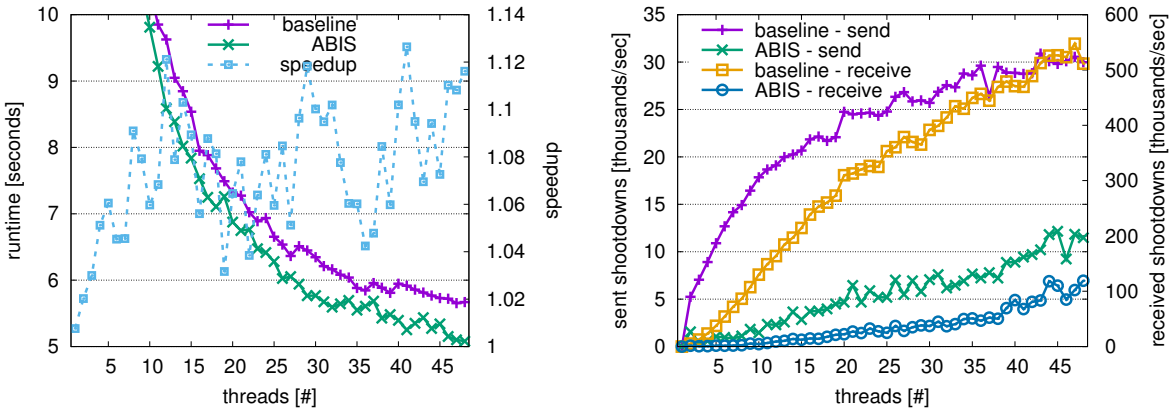
Apache provides several multi-processing modules. We use the default "mpm_event" module, which spawns multiple processes, each of which runs multiple threads. Apache serves each request by creating a memory mapping of the requested file, sending its content and unmapping it. This behavior effectively causes frequent invalidations of short-lived mappings. In the baseline system, the invalidation also requires expensive TLB shootdowns to the cores that run other threads of the Apache process. Effectively, when Apache serves concurrent requests using multiple threads, it triggers a TLB shootdown for each request that it serves.

Figure 6a depicts the number of requests per second that are served when the server runs on different number of cores. ABIS improves performance by 12% when all cores are used. Executing the benchmark reveals that the effect of ABIS on performance is inconsistent when the number of cores is low, as ABIS causes slowdown of up to 8% and speedups of to 42%. Figure 6b presents the number of TLB shootdown that are sent and received in the baseline system and ABIS. As shown, in the baseline system, as more cores are used, the amount of sent TLB shootdowns becomes almost identical to the number of requests that Apache serves. ABIS reduces the number of both sent and received shootdowns by up to 90% as it identifies that PTEs are private and that local invalidation would suffice.

5.3 PBZIP2

Parallel bzip2 (PBZIP2) is a multithreaded implementation of the bzip2 file compressor [20]. In this benchmark we evaluate the effect of reclamation due to memory pressure on PBZIP2, which in itself does not cause many TLB flushes. We use PBZIP2 to compress the Linux 4.4 tar file. We configured the benchmark to read the input file into RAM and split it between processors using 500k block size. We run PBZIP2 in a container and limit its memory to 300MB to induce swap activity. This activity causes the invalidation of long-lived idle mappings as inactive memory is reclaimed.

The time of compression is shown in Figure 7a. ABIS outperforms Linux by up to 12%, and the speedup grows with the number of cores. Figure 7b presents the number of TLB shootdowns per second when this benchmark runs. The baseline Linux system sends nearly 200k shootdowns regardless of the number of threads, and the different shootdown send rate is merely due to the shorter runtime when the number of cores is higher. The number of received shootdowns in the baseline system is



(a) Runtime

(b) Remote shootdowns.

Figure 7: Execution of PBZIP2 when compressing the Linux kernel. The process memory is limited to 300MB to practice page reclamation.

proportional to the number of cores, as the OS cannot determine which TLBs cache the entry, and broadcasts the shootdown messages to all the cores that run the process threads. In contrast, ABIS can usually determine that a single TLB needs to be flushed. When 48 threads are spawned, a shootdown is sent on average to 10 remote cores in ABIS, and to 18 cores using baseline Linux.

5.4 PARSEC Benchmark Suite

We run the PARSEC 3.0 benchmark suite [7], which is composed of multithreaded applications that are intended to represent emerging shared-memory programs. We set up the benchmark suite to use the native dataset and spawn 32 threads. The measured speedup, the runtime, the normalized number of TLB shootdowns and their rate in the baseline system are presented in Figure 8. As shown, ABIS can improve performance by over 3% but can also induce overheads of up to 2.5%. ABIS reduces the number of TLB shootdowns by 96% on average.

The benefit of ABIS appears to be limited by the overhead of the software technique it uses to insert PTEs into the TLB. As this overhead is incurred after each page fault, workloads which trigger considerably more page faults than TLB shootdowns experience slowdown. For example, "canneal" benchmark causes 1.5k TLB shootdowns per second in the baseline system, and ABIS prevents 91% of them. However, since the benchmark triggers over 55k page-faults per second, ABIS reduces performance by 2.5%. In contrast, "dedup" triggers 33k shootdowns and 370k page faults per second correspondingly. ABIS saves 39% of the shootdowns and improves performance by 3%. Hardware enhancements or selective enabling of ABIS could prevent the overheads.

5.5 Limitations

ABIS is not free of limitations. The additional operations and data introduce performance and memory overheads, specifically the insertions of PTEs into the TLB without setting the access-bit. However, relatively simple hardware enhancements could have eliminated most of the overhead (§7). In addition, the CPU incurs overhead of roughly 600 cycles when it sets the access-bit of shared PTEs [37].

To detect short-lived private mappings, our system requires that the TLB be able to accommodate them during their lifetime. New CPUs include rather large TLBs of up to 1536 entries, which may map 6MB of memory. However, non-contiguous or very large working sets may cause TLB pressure, induce evictions, and cause false indications that PTEs are shared. In addition, frequent full TLB flushes, for instance during address-space switching or when the OS sets the CPU to enter deep sleep-state have similar implications. Process migration between cores is also damaging as it causes PTEs to be shared between cores and requires shootdowns. These limitations are often irrelevant to a well-tuned system [30, 31].

Finally, our system relies on micro-architectural behavior of the TLBs. We assume the MMU does not perform involuntary flushes and that the same PTE is not marked as "accessed" multiple times when it is already cached. Experimentally, this is not always the case. We further discuss these limitations in §7.

6. Related Work

Hardware Solutions. The easiest solution from a software point of view is to maintain TLB coherency in hardware. DiDi uses a shared second-level TLB directory that tracks which PTEs are cached by which

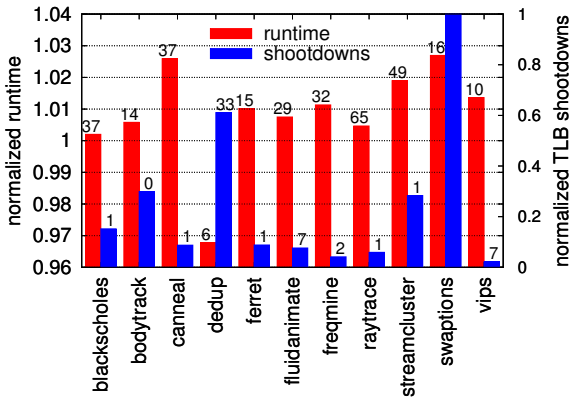


Figure 8: Normalized runtime and number of TLB shootdowns in ABIS when running PARSEC benchmarks. The numbers above the bars indicate the baseline (left) runtime in seconds and (right) rate of TLB shootdowns in thousands/second.

core and performs TLB flushes accordingly [48]. Teller et al. proposed that OSes save a version count for each PTE, to be used by hardware to perform TLB invalidations only when memory is addressed via a stale TLB entry [39]. Li et al. eliminate unwarranted shootdowns of PTEs that are only used by a single core by extending PTEs to accommodate the core that first accessed a page, enhancing the CPU to track whether a PTE is private and avoiding shootdowns accordingly [27].

These studies present compelling evaluation results; however, they require intrusive micro-architecture changes, which CPU vendors are apparently reluctant to introduce, presumably due to a history of TLB bugs [1, 16, 17, 35, 46].

Software Solutions. To avoid unnecessary recurring TLB flushes of invalidated PTEs, Uhlig tracks TLB versions and avoids shootdowns when the remote cores already performed full TLB flushes after the PTE changed [43, 44]. However, the potential of this approach is limited since even when TLB invalidations are batched, the TLB is flushed shortly after the last PTE is modified.

An alternative approach for reducing TLB flushes is to require applications to inform the OS how memory is used or to control TLB flushes explicitly. Corey OS avoids TLB shootdowns of private PTEs by requiring that user applications define which memory ranges are private and which are shared [10]. C4 uses an enhanced Linux version that allows applications to control TLB invalidations [40]. These systems, however, place an additional burden on application writers. Finally, we should note that reducing the number of memory mapping changes, for example by improving the memory reclamation policy, can reduce the number of TLB flushes. However, these solutions are often workload dependent [45].

7. Hardware Support

Although our system saves most of the TLB shootdowns, it does introduce some overheads. Hardware support that would allow privileged OSes to insert PTEs directly to the TLB without setting the access-bit would eliminate most of ABIS's overhead. Such an enhancement should be easy to implement as we achieve an equivalent behavior in software.

ABIS would able to save even more TLB flushes if CPUs avoid setting the PTE access-bit after the PTE is cached in the TLBs. We encountered, however, in situations where such events occur. It appears that when Intel CPUs set the PTE dirty-bit due to write access, they also set the access-bit, even if the PTE is already cached in the TLB. Similarly, before a CPU triggers a page-fault, it performs a page-walk to retrieve the updated PTE from memory and may set the access-bit even if the PTE disallows access. Since x86 CPUs invalidate the PTE immediately after, before invoking the page-fault exception handler, setting the access-bit is unnecessary.

CPUs should not invalidate the TLB unnecessarily, as such invalidations hurt performance regardless of ABIS. ABIS is further affected, as these invalidations cause the the access-bit to be set again when the CPU re-caches the PTE. We found that Intel CPUs (unlike AMD CPUs) may perform full TLB flushes when virtual machines invalidate huge pages that are backed by small host pages.

8. Conclusion

We have presented two new software techniques that prevent TLB shootdowns in common cases, without replicating the mapping structures and without incurring more page-faults. We have shown its benefits in a variety of workloads. While our system introduces overheads in certain cases, these can be reduced by minor CPU enhancements. Our study suggests that providing OSes better control over TLBs may be an efficient and simple way to reduce TLB coherency overheads.

Availability

The source code is publicly available at:
http://nadav.amit.to/publications/tlb.

Acknowledgment

This work could not have been done without the continued support of Dan Tsafrir and Assaf Schuster. I also thank the paper reviewers and the shepherd Jean-Pierre Lozi.

References

[1] Lukasz Anaczkowski. Linux VM workaround for Knights Landing A/D leak. Linux Kernel Mailing List, lkml.org/lkml/2016/6/14/505, 2016.

[2] Manu Awasthi, David W Nellans, Kshitij Sudan, Rajeev Balasubramonian, and Al Davis. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *ACM/IEEE International Conference on Parallel Architecture & Compilation Techniques (PACT)*, pages 319–330, 2010.

[3] Thomas W Barr, Alan L Cox, and Scott Rixner. Translation caching: skip, don't walk (the page table). In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 48–59, 2010.

[4] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D Hill, and Michael M Swift. Efficient virtual memory for big memory servers. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 237–248, 2013.

[5] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–44, 2009.

[6] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. Shared last-level TLBs for chip multiprocessors. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 62–63, 2011.

[7] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *ACM/IEEE International Conference on Parallel Architecture & Compilation Techniques (PACT)*, pages 72–81, 2008.

[8] David L Black, Richard F Rashid, David B Golub, Charles R Hill, and Robert V Baron. Translation lookaside buffer consistency: a software approach. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 113–122, 1989.

[9] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel, Third Edition*. O'Reilly & Associates, Inc., 2005.

[10] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yue-hua Dai, et al. Corey: An operating system for many cores. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 43–57, 2008.

[11] Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, pages 211–224, 2013.

[12] 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. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 105–118, 2011.

[13] Jonathan Corbet. Realtime and interrupt latency. LWN.net, https://lwn.net/Articles/139784/, 2005.

[14] Jonathan Corbet. Memory compaction. LWN.net, https://lwn.net/Articles/368869/, 2010.

[15] Jonathan Corbet. Memory management locking. LWN.net, https://lwn.net/Articles/591978/, 2014.

[16] Christopher Covington. arm64: Work around Falkor erratum 1003. Linux Kernel Mailing List, https://lkml.org/lkml/2016/12/29/267, 2016.

[17] Linux Kernel Driver DataBase. CONFIG_ARM_ERRATA_720789. http://cateee.net/lkddb/web-lkddb/ARM_ERRATA_720789.html, 2012.

[18] Jake Edge. Persistent memory. LWN.net, https://lwn.net/Articles/591779/, 2014.

[19] Balazs Gerofi, Akira Shimada, Atsushi Hori, and Yozo Ishikawa. Partially separated page tables for efficient operating system assisted hierarchical memory management on heterogeneous architectures. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 360–368, 2013.

[20] Jeff Gilchrist. Parallel data compression with bzip2. In *IASTED International Conference on Parallel and Distributed Computing and Systems (ICPDCS)*, volume 16, pages 559–564, 2004.

[21] Mel Gorman. TLB flush multiple pages per IPI v4. Linux Kernel Mailing List, https://lkml.org/lkml/2015/4/25/125, 2015.

[22] Julien Grall. Force broadcast of TLB and instruction cache maintenance instructions. Xen development mailing list https://patchwork.kernel.org/patch/8955801/, 2016.

[23] Dave Hansen. Patch: x86: set TLB flush tunable to sane value. https://patchwork.kernel.org/patch/4460841/, 2014.

[24] Xeon phi processor. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

[25] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual. Reference number: 325462-057US, 2015. https://software.intel.com/en-us/articles/intel-sdm.

[26] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with Ingens. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 705–721, 2016.

[27] Yong Li, Rami Melhem, and Alex K Jones. PS-TLB: Leveraging page classification information for fast, scalable and efficient translation for future CMPs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):28, 2013.

[28] Likai Liu. Parallel computing and the cost of TLB shoot-down. http://lifecs.likai.org/2010/06/parallel-computing-and-cost-of-tlb.html, 2010.

[29] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. TLB improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level TLBs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(1), 2013.

[30] Ophir Maor. What is CPU affinity? https://community.mellanox.com/docs/DOC-1924, 2014.

[31] Ophir Maor. Mellanox BIOS performance tuning example. https://community.mellanox.com/docs/DOC-2297, 2015.

[32] Marshall Kirk McKusick, George V Neville-Neil, and Robert NM Watson. *The design and implementation of the FreeBSD operating system*. Pearson Education, 2014.

[33] Jinzhan Peng, Guei-Yuan Lueh, Gansha Wu, Xiaogang Gou, and Ryan Rakvic. A comprehensive study of hardware/software approaches to improve TLB performance for Java applications on embedded systems. In *ACM Workshop on Memory System Performance and Correctness (MSPC)*, pages 102–111, 2006.

[34] Aristeu Rozanski. VM-scalability benchmark suite. https://github.com/aristeu/vm-scalability, 2010.

[35] Anand Lal Shimpi. AMD's B3 stepping Phenom previewed, TLB hardware fix tested. AnandTech http://www.anandtech.com/show/2477/2, 2008.

[36] Kirill A. Shutemov. mm: map few pages around fault address if they are in page cache. Linux Kernel Mailing List, https://lwn.net/Articles/588802, 2014.

[37] Kirill A. Shutemov. unixbench.score -6.3% regression. Linux Kernel Mailing List, http://lkml.kernel.org/r/20160613125248.GA30109@black.fi.intel.com, 2016.

[38] Patricia J Teller. Translation-lookaside buffer consistency. *IEEE Computer*, 23(6):26–36, June 1990.

[39] Patricia J Teller, Richard Kenner, and Marc Snir. *TLB consistency on highly-parallel shared-memory multiprocessors*. Courant Inst. of Math. Sci, 1987.

[40] Gil Tene, Balaji Iyengar, and Michael Wolf. C4: The continuously concurrent compacting collector. *ACM International Symposium on Memory Management (ISMM)*, pages 79–88, 2011.

[41] Michael Y Thompson, JM Barton, TA Jermoluk, and JC Wagner. Translation lookaside buffer synchronization in a multiprocessor system. In *USENIX Winter*, pages 297–302, 1988.

[42] Linus Torvalds. Splice: fix race with page invalidation. http://yarchive.net/comp/linux/zero-copy.html, 2008.

[43] Volkmar Uhlig. *Scalability of microkernel-based systems*. PhD thesis, TH Karlsruhe, 2005. https://os.itec.kit.edu/downloads/publ_2005_uhlig_scalability_phd-thesis.pdf.

[44] Volkmar Uhlig. The mechanics of in-kernel synchronization for a scalable microkernel. *ACM SIGOPS Operating Systems Review (OSR)*, 41(4):49–58, 2007.

[45] Ahsen J Uppal and Mitesh R Meswani. Towards workload-aware page cache replacement policies for hybrid memories. In *International Symposium on Memory Systems (MEMSYS)*, pages 206–219, 2015.

[46] Theo Valich. Intel explains the Core 2 CPU errata. The Inquirer http://www.theinquirer.net/inquirer/news/1031406/intel-explains-core-cpu-errata, 2007.

[47] Brian Van Essen, Henry Hsieh, Sasha Ames, and Maya Gokhale. DI-MMAP: A high performance memory-map runtime for data-intensive applications. In *IEEE International Workshop on Data-Intensive Scalable Computing Systems (SCC)*, pages 731–735, 2012.

[48] Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S Unsal. DiDi: Mitigating the performance impact of TLB shootdowns using a shared TLB directory. In *ACM/IEEE International Conference on Parallel Architecture & Compilation Techniques (PACT)*, pages 340–349, 2011.

[49] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, volume 36, pages 181–194, 2002.

[50] wrk. HTTP benchmarking tool. https://github.com/wg/wrk, 2015.

Falcon: Scaling IO Performance in Multi-SSD Volumes

Pradeep Kumar H. Howie Huang
The George Washington University

Abstract

With the high throughput offered by solid-state drives (SSDs), multi-SSD volumes have become an attractive storage solution for big data applications. Unfortunately, the IO stack in current operating systems imposes a number of *volume-level* limitations, such as per-volume based IO processing in the block layer, single flush thread per volume for buffer cache management, locks for parallel IOs on a file, all of which lower the performance that could otherwise be achieved on multi-SSD volumes. To address this problem, we propose a new design of *per-drive IO processing* that separates two key functionalities of IO batching and IO serving in the IO stack. Specifically, we design and develop *Falcon*[1] that consists of two major components: *Falcon IO Management Layer* that batches the incoming IOs at the volume level, and *Falcon Block Layer* that parallelizes IO serving on the SSD level in a new block layer. Compared to the current practice, Falcon significantly speeds up direct random file read and write on an 8-SSD volume by 1.77× and 1.59× respectively, and also shows strong scalability across different numbers of drives and various storage controllers. In addition, Falcon improves the performance of a variety of applications by 1.69×.

1 Introduction

The demand of high-performance storage systems is propelled by big data applications that need high IO throughput for processing massive data volumes. Flash-based solid-state drives (SSDs) provide an attractive option compared to hard disk drives for such applications, due to their high random and sequential performance. As a common practice, multiple SSDs are increasingly deployed to support a wide variety of applications such as graph analytics [23, 50, 20, 51, 31, 26], machine-learning [21, 30], and key-value stores [11, 25]. In this work, we especially use a number of graph analytics systems as motivating examples to illustrate the drawbacks of existing approaches.

[1]This system is named after the Millennium Falcon in the Star Wars, "the fastest ship in the galaxy".

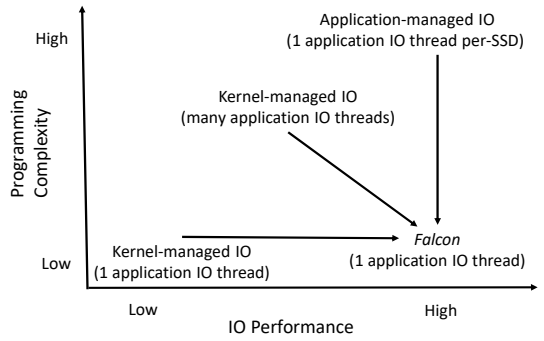


Figure 1: Falcon aims to achieve both ease of programing and high IO performance.

To take advantage of high performance of SSDs, throughput-sensitive applications either utilize an application-managed or kernel-managed IO approach as illustrated in Figure 1. In the first case of application-managed IO, prior projects such as SAFS [49], FlashGraph [50], and Graphene [23] require the application developer to explicitly distribute the data among multiple files, each hosted in independent SSDs. In this case, there is no abstraction of a volume, and one application IO thread is dedicated to each SSD. Clearly, such a framework is very complex as applications need to be aware of data partitioning, and determine which application IO thread should perform the IO at any particular instance.

On the other hand, for kernel-managed IO, applications can enjoy the benefits of both volumes and batched IO interfaces provided by the operating system, e.g., Linux AIO (asynchronous IO), Solaris KAIO and Windows Overlapped IO. Such interface allows the applications to submit multiple IOs within a single system call, which provides a clear advantage of ease of programming. However, because IO functionality is limited to just one application IO thread, the combination of kernel-managed IO and a volume, be it created by Linux (e.g., *md*, *lvm*), FreeBSD (e.g., *geom*) or hardware RAID, would fail to saturate the aggregate bandwidth of multiple SSDs.

To mitigate this problem, the applications can spawn a number of dedicated application IO threads to serve the requests in parallel. Several existing projects adapt this

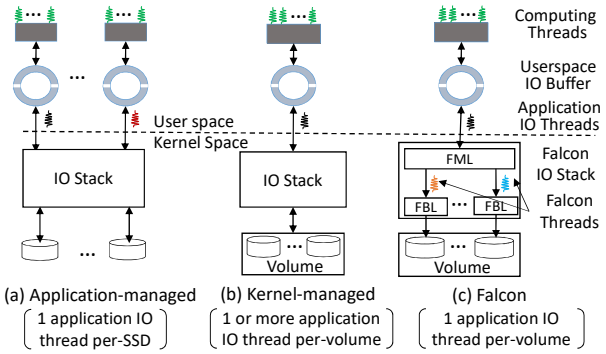


Figure 2: (a) Applications explicitly manage data and IOs on each SSD. There is no volume; (b) Applications relies on the kernel to manage the data on the volume; (c) Falcon maintains the abstraction of volumes and utilizes specialized kernel threads, called Falcon threads, to parallelize per-drive processing for high throughput.

approach, including GridGraph [51] and G-Store [20]. Unfortunately, managing multiple application IO threads using a thread pool is again complicated. And in many cases, this approach does not achieve the expected goal due to the limitations in IO subsystems [28]. For example, many file systems (e.g., ext4 [10]) apply a per-file inode lock, which prevents scalable random read or write from a single file, irrespective of how many application IO threads are employed. So is the case for buffered write where a single kernel thread per volume is responsible for flushing the dirty buffer cache to the volume, limiting the write throughput that could potentially be achieved.

In this work, we strive to achieve the combined benefits of both approaches, that is, delivering high performance IO on a multi-SSD volume while providing ease of programming to the application developers. To this end, we design and develop *Falcon* whose workflow, as shown in Figure 2, presents a new design of *per-drive IO processing* on multi-SSD volumes. The key insight is the separation of the two functionalities of *IO batching* and *IO serving* in the IO stack. The former batches and classifies the incoming IOs at the volume level, and is performed in Falcon IO Management Layer (FML). Meanwhile, the latter serves the IOs in parallel to the SSDs, and is performed in Falcon Block Layer (FBL) using a specialized kernel thread, called Falcon thread.

In particular, FBL provides two new techniques: (1) *per-drive IO sort and neighbor merge*, which limits the scope of sort operations to each SSD and merge to neighboring requests. In contrast, the Linux IO merge algorithm unnecessarily traverses every IO request for all the member SSDs. And (2) *dynamic tag allocation*, which assigns request tags, a limited hardware resource, at runtime. This helps to reduce the unpredictable blocking in the IO stack, and provide a better mechanism to control the number of active IOs in the pipeline, which is applicable across different storage technologies and vendors.

As a result, Falcon allows a dedicated application IO thread to saturate the multi-SSD volume. Thus developers can concentrate more on algorithmic optimizations, without worrying about the complexity of managing multiple application IO threads and SSDs. In contrast, Linux follows *per-volume* approach of mixing IO batching and IO serving tasks in the block layer, where the sequential IO processing and round-robin dispatch lead to many inefficiencies on multi-SSD volumes, and limit the parallelism that could otherwise be achieved.

We have evaluated Falcon with a number of micro-benchmarks, real applications, and server traces. Falcon shows strong scalability across different numbers of SSDs, and several different storage controllers. On an 8-SSD volume, Falcon significantly speeds up direct random read and write throughput on an ext4 file by 1.77× and 1.59× respectively, buffered random write by 1.59×, and shows consistent performance for various stripe size configurations. In addition, Falcon speeds up graph processing, utility applications, filebench and trace replay by 1.69×. Lastly, it is important to note that with the new block layer, Falcon is able to saturate a non-volatile-memory-express (NVMe) SSD, delivering 1.13× speedup over the native Linux.

The remainder of the paper is organized as follows. Section 2 presents background on volume management and its interaction with the block layer, as well as how an IO request traverses through various layers. Section 3 quantifies the challenges arising due to per-volume philosophy of Linux IO stack, and presents an overview of Falcon architecture. Section 4 and 5 present the design and implementation of Falcon components. We evaluate the performance of our techniques in Section 6, discuss related works in Section 7, and conclude in Section 8.

2 Background

In this work, while we mostly use Linux to describe the background on the volume management and the block layer; it is worth noting that this IO workflow is generic in nature and many operating systems implement a similar mechanism. Nevertheless, our design and implementation have been influenced by Linux-specific techniques.

In particular, we compare to the Blk-mq [1] block layer which has shown better scaling than the single-queue block layer. Also, most of our discussions pertain to single application IO thread using batched IO interfaces such as Linux AIO. Many kernel daemons such as *pdflush* and *kjournald* submit IO internally in a way similar to batched IO interface. Specifically, pdflush daemon manages the page cache, and has only one dedicated kernel thread per volume to write the dirty pages to storage. There is no pdflush thread to manage the read, and it hap-

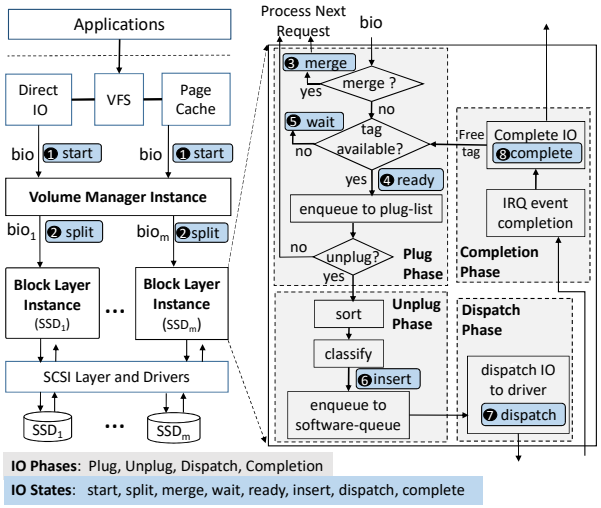


Figure 3: Left: Linux IO stack and the interaction between the volume manager and the block layer. Right: the block layer instance and the detailed IO flow. IO processing happens sequentially, while dispatch happens in a round-robin fashion.

pens directly in the context of the IO issuing thread. Figure 3 shows the inner working of a multi-SSD volume.

Volume Management and Block Layer. An instance of the block layer is associated to a block device, which is associated to a single drive such as an SSD. The volume management layer is used to map several physical block devices into a single virtual block device (e.g., md or lvm). In this layer, IO requests are represented as an object of *block IO* (bio for short) structure. The job of the volume manager instance is to break the incoming bio object into multiple (smaller) bio objects destined for member drives, depending on the IO size and the stripe size of the volume, as discussed next. The original IO is completed only when all the split IOs to different drives are completed.

IO Flow and States. Figure 3 also shows the flow of an IO request from submission to completion within the block layer. For simplicity, we group the process to four phases: *plug*, *unplug*, *dispatch*, and *completion*. IO batching, merge, and tag allocation are performed in the plug phase. IO batching provides an opportunity to merge incoming IOs to take advantage of higher sequential throughput. Also, SSDs provide higher throughput for batched IOs due to parallelism at the hardware level, where more than one IO can be fetched in parallel. Next, sort and classify operations are performed in the unplug phase, IO requests are dispatched to SSDs in the dispatch phase, and IO completion is performed in the the completion phase where various resources are freed.

In each phase, the IO request advances across various states as different tasks are performed on it. As we will show later, one may use the states to track the IO, and find out the time spent by an IO request in different phases for performance profiling.

As soon as an IO request enters the kernel, it is converted to a *struct bio* object and assumes the ❶*start* state. In the case of a multi-SSD volume, the volume manager splits the bio object into multiple smaller objects and moves them to the ❷*split* state. For example, for a multi-SSD volume of 4KB stripes, an incoming IO request of 64KB would be divided into 16 bio objects, each containing 4KB IO destined to a specific SSD. Next, a number of block layer instances (one per SSD) handle the incoming IOs as if it were an IO to this particular SSD. For example, in Figure 3, *bio*1 proceeds to the block layer instance of SSD1, *bio*2 to SSD2, and so forth.

These split bio objects enter their block layer instances in a sequential fashion, and the *plug* phase begins. The operation starts with the bio object being checked against existing IO entries of the per-core *plug-list* for merge candidates. As illustrated in Figure 4(a), the plug-list is a private queue to each IO thread, and is used for batching and merging the incoming IO requests. In other words, the plug-list is shared among multiple block layer instances, and used by all member SSDs of the multi-SSD volume. In this case, a thread does sequential processing of all previously split bio objects, and presents several drawbacks, as we will discuss shortly.

If the bio object is merged, then it moves to the ❸*merge* state, and the processing of the next object starts. Otherwise, a request tag will be requested. If a tag is available then the bio object is put inside a unique *struct request* container indexed by the allocated tag, which in turn is queued to the plug-list. This state is called the ❹*ready* state as IO requests are dispatched in this form to the physical drivers later. However, if a request tag were not available, the IO moves to the ❺*wait* state, and the thread blocks waiting for a free tag.

When the number of IO requests in the plug-list reaches a threshold, an *unplug* event happens, and the *unplug* phase starts. In this phase, all the IOs present in the plug-list of this thread are sorted based on the destination drive and block address information. Next, the sorted IOs move to the per-core, per-drive *software queue* of the member drives, and acquire the ❻*insert* state.

In the *dispatch* phase, the IO requests are dispatched in a round-robin fashion from the software queues to the drives in the same thread context, and moves the IOs to the ❼*dispatch* state. If some IOs can not be dispatched, they will be kept in the per-drive *dispatch-queue* (not shown in the Figure 3) for later processing. Lastly in the *completion* phase, when a drive completes an IO, it raises an IRQ event. The IRQ handler will free the resources and move the IO request to the ❽*complete* state, where any waiting thread is woken up.

Request Tag. The request tag is a limited, vendor and technology specific hardware resource [9]. The available tags are either per storage controller or per-drive.

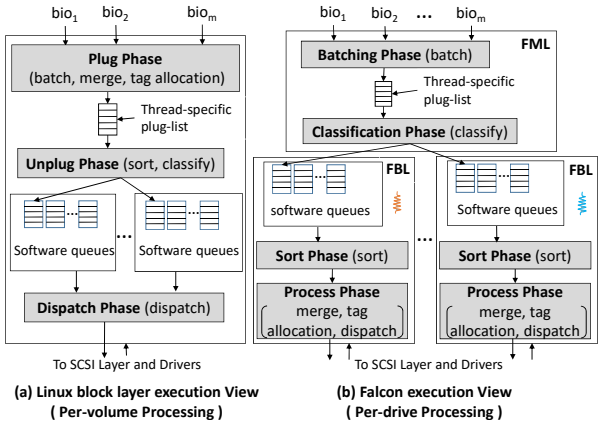


Figure 4: (a) IO issuing thread batches the IOs destined to different member drive to same plug-list. The merge, tag allocation, and sort operation being performed in the plug-list is the major cause of inefficiency. (b) Falcon's idea of per-drive philosophy is to postpone the IO serving tasks of the block layer to the drive-specific software-queue. Completion phase is omitted for simplicity.

For example, the Intel SCU technology has 250 available tags [39], but they are shared among four ports of the controller. That is, every connected SSDs will compete for the same tag space. Similarly, the LSI 9300-8i SAS HBA adapter has 10,140 tags, and is shared by all the connected drives. On the other hand, the Intel AHCI SATA controller has only 32 tags per SATA port, which is not shared. In this case, the tag count matches with the drive's internal queue size. For the Samsung 950 pro 512GB NVMe SSD that we use in this work, the tag counts are 1024 per hardware-queue. This specific drive has 8 hardware queues [38], while SATA SSDs have only one hardware queue.

3 Falcon Architecture

In this section, we first describe the insufficiencies of current per-volume processing of Linux IO stack, and present the overall architecture of Falcon.

3.1 Challenges of Per-Volume Processing

Current multi-SSD volumes follow the per-volume processing, that is, IO serving is tied to the plug-list, and is forced to be performed in a sequential manner within a volume. In other words, as shown in Figure 4(a), the plug-list mixes IO requests that actually belong to various member drives within a multi-SSD volume. Moreover, the block layer mingles two unrelated tasks: batching, and merge/tag allocation in its plug phase, and sort and classify in the unplug phase.

To illustrate the problems, we run a revised FIO benchmark [12] on various configurations of multi-SSD volumes. The detailed setup will be presented in Section 6. In particular, we measure two metrics: the *stack latency* is the time between the start and insert states, while the *device latency* is the time between the dispatch and completion states. We use the former to gauge the software performance, and the latter for device performance.

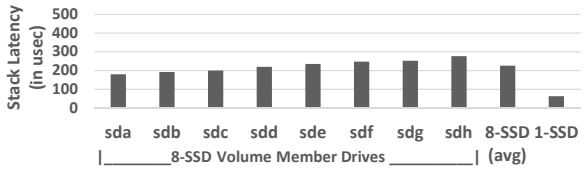


Figure 5: Stack latency of eight SSDs, and average latency

Insufficiency #1: Lack of Parallelism. As several IO serving tasks are forced to be performed in a single plug-list, the opportunities in parallelizing those tasks are limited. Under the Linux architecture shown in Figure 4(a), merge, tag allocation, and sort tasks lack parallelism, while dispatch happens in a round-robin way.

Figure 5 quantifies this impact on the stack latency of SSDs within a volume. Out of 8 SSDs, the slowest drive (*sdh*) spends at least 55% more time on IO processing (i.e., the stack latency) as compared to the fastest drive (*sda*). Interestingly, the latency increases in the same order of the drives. This is due to the round-robin dispatch where the first drive always gets the highest priority to dispatch followed by the second drive onwards. As a result of this procedure, later drives have to wait even though the requests are ready to be dispatched.

Insufficiency #2: Inefficient Merge and Sort. The current merge algorithm traverses the plug-list of the thread to find the merge candidate for a bio object. It searches all IO requests including those that belong to different drives. But clearly, they should not be considered as candidates at all. Also, in the unplug phase, sorting happens on the same thread-specific plug-list, which again means wasteful processing on irrelevant requests.

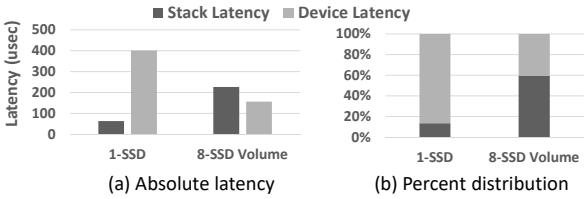


Figure 6: Distribution of the stack latency and device latency, showing absolute and percentage distribution. For 8-SSD volume, stack latency is more than the device latency.

Making matters worse, the IO count in the plug-list is significantly higher for a multi-SSD volume. The plug phase ends only when the merge task finds more than 16 IOs (the per-drive threshold) in the plug-list destined to the same drive. Assuming an uniform distribution of the IOs among all the drives, the total number of the IOs in the plug-list would need to reach 128 for an 8-SSD volume to end the plug phase, as opposed to 16 for 1-

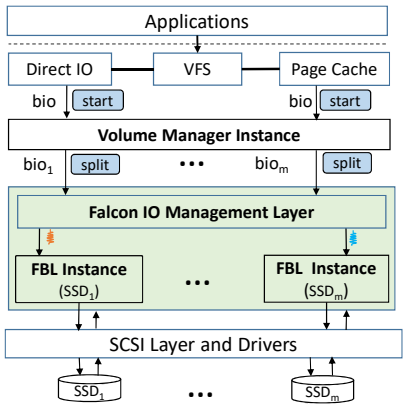


Figure 7: Falcon IO stack highlighting differences with Linux IO stack. A new abstraction in the form of *Falcon IO Management Layer* (FML) is introduced which performs purely IO batching work, while the Falcon Block Layer (FBL) performs IO serving tasks in parallel to FML and to each other.

SSD. As a result, the average processing time spent by an IO thread for the 8-SSD volume is significantly higher, over 3.5× more than 1-SSD (Figure 6(a)), and forces the IO thread to spend 60% time inside the IO stack, pointing to the IO stack as the bottleneck (Figure 6(b)).

Insufficiency #3: Unpredictable Blocking. In between the merge and sort tasks, the tag allocation is forced to be performed in a sequential manner as well. So, when a tag allocation fails for any drive member, the executing thread blocks the whole IO stack waiting for a free tag from that drive. Thus the active IO count present in the Linux IO stack is controlled by the tag count because the blocked IO thread wakes up only when the tag becomes available, i.e. only when an existing IO completes. This blocking is unpredictable, as the tag count varies and can either be storage controller or drive specific.

3.2 Per-Drive Processing in Falcon

Falcon proposes a new approach of the *per-drive philosophy*, which separates the two operations of IO batching and IO serving by regrouping the tasks by their functionalities in new phases, as shown in Figure 4(b). Specifically, only IO batching and classify tasks are performed in the plug-list. And merge, tag allocation, and dispatch tasks move to a new *process phase* and are performed in per-drive software queues, and can easily be parallelized. This reduces the amount of work being done in the plug-list, and hence removes the major bottlenecks.

Figure 7 presents the major components of Falcon. In particular, the new batching and classification phases are performed in the *Falcon IO Management Layer* (FML for short), while the sort phase along with the process and completion phases are performed in the *Falcon Block Layer* (FBL). Moreover, the FML also spawns Falcon threads for parallel IO serving across FBL instances.

Block Layer Features	Linux 1-SSD	Linux Multi-SSD	Falcon Volume
Parallel processing	NA	✗	✓
Per-drive sort	✓	✗	✓
Neighbor merge	✗	✗	✓
Dynamic tag management	✗	✗	✓

Table 1: Falcon's per-drive processing

We summarize the differences between Falcon and Linux IO stack in Table 1. For example, the Linux Blk-mq architecture allows per-drive sort for 1-SSD system, but it fails to provide the same functionality to multi-SSD volumes. In contrast, the separation of functionalities allows the Falcon to keep the per-drive philosophy intact in its FBL block layer, as the IO serving operations are performed in the per-drive software queue. In addition, dynamic tag allocation in FBL removes the tag allocation from the plug phase and moves it to just before dispatch. This provides a more uniform and predictable criterion to control the outstanding IOs in the IO stack pipeline.

4 Falcon IO Management Layer

Falcon IO Management Layer (FML) is the new abstraction between the volume manager and the block layer. It performs IO batching, and creates a Falcon thread for each FBL instance to parallelize per-drive processing. Figure 8 presents the IO flow in the management layer.

4.1 IO Batching

IO batching is performed in two phases: batching and classification. FML starts its batching phase as soon as a new (split) bio arrives, and pushes the object into the plug-list of the thread. Next, the volume manager sends the next bio object of the original IO request. If there are no more objects, the volume manager processes the next request from the batched IO interface. This bio object is again enqueued to the same plug-list by the FML layer.

As this process progresses, an unplug event will occur. At this point, the current batching phase stops, and the classification phase begins, thereby, all the bio entries in the plug-list are classified based on destination drives.

Batching and classification tasks are performed using bio objects in Falcon, as opposed to request containers in Linux, hence we add *prev* and *next* pointers to the bio structure, so that the bio objects can be chained in the doubly linked-list plug-list.

Additionally, at the end of the classification phase, all the IOs have to be enqueued to the per-core per-drive software queue. However, software queues are protected by spin locks, and acquiring them becomes mandatory each time an IO need to be enqueued. To this end, we collect the requests in temporary per-drive queues during the classify operation. At the end, all bio objects from the

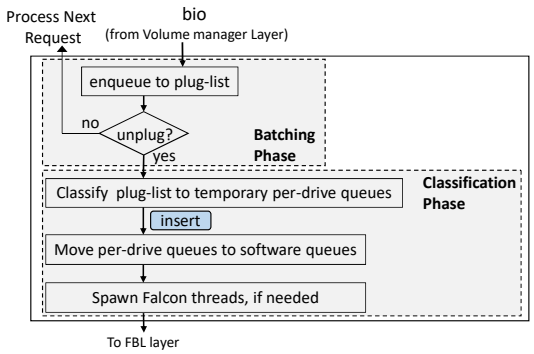


Figure 8: IO flow in Falcon IO Management Layer. The plug-list is used only for batching operations. Merge, sort, and tag operations are no longer performed in here.

temporary queues move to respective software queues, thus acquiring the spin lock only once.

4.2 Enabling Parallel Processing

At the end of the classification phase, all the IOs reside in the *per-drive* software queue, and hence the FBL instances can perform IO serving tasks in parallel, which is not possible in Linux. To this end, FML spawns one kernel thread per participating FBL instance for this group of batched IOs. We call each thread a *Falcon thread*, and is responsible for IO serving tasks.

In our implementation, the Falcon threads are spawned using Linux's kblockd workqueue object. However, the CPU affinity of a kworker is decided by the thread that requests a kworker. Should all the Falcon threads share the same core as the IO issuing thread, it would defeat the purpose of parallel IO serving. To address this problem, we modify the workqueue API invocations so that we can use the CPU affinity of each Falcon thread to pin them to different cores. Ideally it should be NUMA-aware, that is, on a core of the socket that hosts the corresponding storage controller adapter. We manage this information inside *hardware context* object of each drive.

The completion phase is also executed in the same core to which the Falcon thread is pinned. As the CPU core information is now available, the IRQ handler can send an inter-processor interrupt (IPI) to this core to perform the completion phase, or execute directly if the IRQ is received on the same core as that of the Falcon thread.

The job of completion phase is three-fold: freeing up bio objects, request tag and other resources; waking up any thread that is waiting for IO completion; and requesting a Falcon thread to resume processing if the internal queue of the drive becomes full at the last dispatch. In performing those tasks, a completion thread accesses those data structures that are set by the Falcon thread. Hence by running the completion on the same core, Falcon avoids the cache migration of those objects, making completion a cache friendly phase.

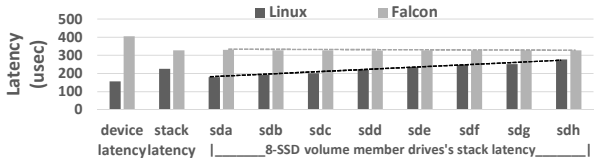


Figure 9: Drive and stack latency comparison of 8-SSD volume. Falcon is able to make the stack latency uniform as well as smaller than the device latency.

4.3 Unplug Criteria

One implication of separating the merge task from the batching task is removal of the unplug criteria from the plug phase. The unplug event is raised when the number of requests evaluated for merging goes beyond a threshold in the plug phase. As we have mentioned earlier, the criteria varies depending on the IO distribution to each drive, and the maximum value of the threshold is 128 for 8-SSD volume in a uniform distribution. To achieve more predictable unplug events, Falcon utilizes two new thresholds (*low watermark* and *high watermark*). Simply put, we maintain the IO requests in the plug-list and finish the batching phase when the count reaches beyond a threshold.

Increasing the watermark by too much would increase the stack latency, and as a result the device would remain idle because more IO requests are still being processed. On the other hand, lowering the watermark may potentially reduce the benefit of batching. An equilibrium is desired so that the drives are kept busy as long as there are sufficient number of IO requests. Given a typical internal queue size of 32 for an SSD, we choose the product of device count and this queue size as the high watermark value, e.g., 256 for an 8-SSD volume.

Figure 9 plots the stack latency for Falcon using the high watermark. Since the ordering of IO state has changed, here we measure the new stack latency as the time between the start to ready phase. One can see that Falcon is able to achieve similar stack latency for different drives, compared to a large variance in Linux. Specifically, Falcon achieves around 320 microseconds, smaller than 404.7 microseconds device latency from the SSDs. As such, the stack and drive latency are nicely balanced.

It should be noted that the device latency is a function of the queue depth, i.e. a busy SSD will have higher device latency than one that is lightly loaded. In Linux's case, the device is operating at lower queue depth, as the application IO thread is not able to dispatch enough requests. In contrast, thanks to parallelism, Falcon threads in Falcon are able to dispatch more IOs to SSDs, and keep them busy all the time. As a result, the stack latency is smaller than the device latency. So, even with higher stack latency than Linux, Falcon is able to get higher throughput as we will show in Section 6.

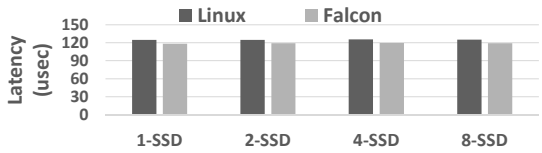


Figure 10: IO latency study of various multi-SSD volume

Latency. Falcon uses the low watermark to facilitate latency-sensitive applications, where only fewer IOs are submitted. The idea is to avoid an extra context switch when the IO demand is low. Here we take current value of 16 requests per drive as the basis, and set the low watermark as the product of this value and the drive count, e.g., 128 IOs for 8-SSD volume. It should be noted that if fewer IOs are submitted in a batched IO interface or just one IO using POSIX IO interface, the batching phase does not wait for more incoming IOs, and an unplug event occurs at the end of the submission. For 1-SSD system, Falcon always performs synchronous IO serving. Note that it is also possible to let the users choose both high and low watermarks depending on their need.

For such applications, IO serving will happen in the context of the IO issuing thread as the plug-list would not cross the low watermark. Figure 10 shows that Falcon improves the IO latency (from the application perspective) by nominal 3% for various multi-SSD volumes (RAID0, 4KB stripe size) when just one IO of size 4KB (POSIX IO) is active in the whole IO pipeline.

5 Falcon Block Layer

Falcon Block Layer (FBL) is the new block layer that performs the IO serving tasks. FBL instances receive unsorted bio objects in their per-core, per-drive software queues. Compared to the existing approach where most of the operations happen in the per-thread plug-list, our approach enables per-drive processing, which can be divided into three phases (sort, process, and completion) as shown in Figure 11.

5.1 Per-Drive Sort and Neighbor Merge

Mechanism. The software queue is a per-core queue, so a single drive has many associated software queues, one for each CPU core. Hence the sort phase first aggregates the bio objects from all of the software queues of the drive in a private queue (a doubly linked-list), and then performs sorting on it. This results in all neighboring IOs being adjacent to each other in the private queue, thus only a neighbor merge is required in the process phase, which happens as follows.

A Falcon thread removes the first bio object from the private queue, allocates a tag, and puts it inside a request container object indexed by the tag. Then, the merge task checks the next bio entry in the private queue to see if it

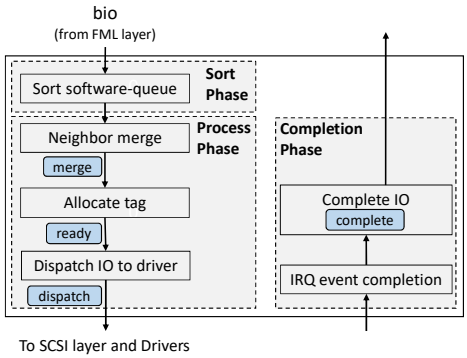


Figure 11: Falcon Block Layer IO flow, and states

can merge with the current container. If it succeeds, the next entry in the queue will be tried for merge. Otherwise, the container object will be dispatched. The process goes on till either all entries are dispatched or the internal queue of the drive becomes full.

The internal queue of an SSD may become full due to its limited size. In this case, all the requests need to be preserved to be dispatched later. The last request container is kept in the *dispatch-queue*. We introduce a new per-drive queue, called *bio-queue*. Its role is similar to dispatch-queue, but keeps the remaining bio entries of the private queue. Later, when triggered, IOs are first dispatched from the dispatch-queue followed by the bio-queue. The separation of IOs in different queues are required as the IOs are in different states. The order maintains the prior behavior of request dispatch.

The sort task requires multiple pass over IOs in the private queue which collects bio objects from software queues. It is possible that the sort task might dominate the overall processing in the IO stack. We leave the investigation of a new data-structure for queues as future work.

Advantages. The new merge technique is very simple and presents several benefits. First, sorting runs efficiently with less CPU usage due to smaller per-drive sort space. Second, the merge algorithm needs to evaluate its neighbor requests only as they are already sorted, which reduces the CPU utilization further. Third, since merge happens on the private queue containing IOs from software queues of the different cores, one can automatically achieve merging across multiple IO issuing threads.

It is worth noting that efficient sort and neighbor merge are generic improvements to Linux IO stack. For example, single application IO thread is not able to saturate an NVMe SSD (Samsung 950 pro 512GB NVMe) in Linux [18]. However, as shown in Figure 12, Falcon can saturate it (1375 MB/s) for random read workload using FIO benchmark. In this case, Falcon does synchronous IO serving by default.

Linux Blk-mq layer treats NVMe SSDs differently from SATA SSDs. Only two incoming IOs are consid-

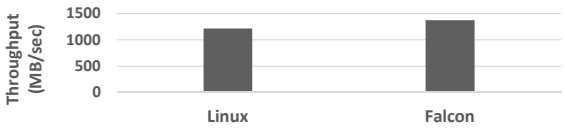


Figure 12: Impact on NVMe SSD for random read

ered for merging for NVMe SSDs, and if that fails, the older IO is dispatched and the most recent is kept in the plug-list. In contrast, Falcon does not differentiate between NVMe and SATA SSDs. In this case, per-drive sort and neighbor merge makes the batching efficient, without any sacrifice on latency.

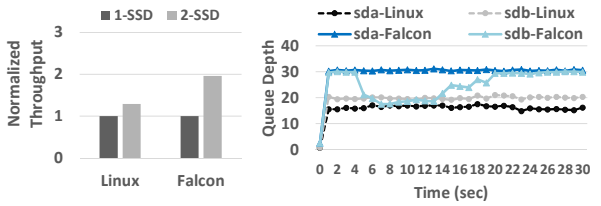
5.2 Tag Management

Problems. Controlling the active IO count in the IO pipeline based on a vendor and technology specific tag count often leads to unpredictable results for random IOs in multi-SSD volumes. Random IOs are inherently skewed towards some drives within any small time duration. This leads to unfairness in the tag allocation for member drives, resulting in compromised performance for Intel C602 AHCI SATA III connected volume as shown in Figure 13(a).

The reason is that after allocating 32 tags for a SATA SSD, the IO thread would block for an additional tag for the SSD, even if other SSDs might have available tags. In a skewed IO distribution case, 2-SSD SATA volume can only maintain less than 40 active IOs in the IO pipeline against the available tag of 64 as shown in Figure 13(b), resulting in the throughput drop. When there are sufficient number of tags such as LSI HBA which has over 10,000 tags, the volume does scale on multiple SSDs on both Linux and Falcon.

Dynamic Tag Allocation. To provide a predictable behavior, a uniform count of active IOs must be maintained in the IO pipeline, regardless of the storage technology or vendor. Therefore, Falcon performs the tag allocation *dynamically*, i.e. only if a dispatch is required in the process phase, as shown in Figure 11. The main benefit is improved queue utilization because more IOs are allowed to reside in the IO pipeline without acquiring a tag. This offsets the skewness of random IO distribution.

Figure 13(a) compares the throughput scaling of Linux and Falcon for a 2-SSD volume connected using Intel C602 AHCI SATA III. The throughput improvement is due to improved tag utilization of both the member drives as shown in Figure 13(b). The drop in tag usage between 5–19 seconds is due to highly skewed workload distribution (random read in FIO benchmark), where only one drive's internal queue is fully utilized. However, Falcon can still get close to 2× IO performance improvement over 1-SSD volume. The technique results in 52% and 23% improvement in random read and write respectively, saturating the volume completely.



(a) IO Throughput Scaling (b) Tag usage in 2-SSD SATA volume

Figure 13: Impact of dynamic tag allocation on 2-SSD SATA volume connected using the Intel C602 controller

Back Pressure. Tag allocation serves as a back pressure point in the Linux IO stack. That is, if the number of in-flight IOs were to increase beyond the available tags, the IO issuing thread would go to sleep and stop submitting new IOs. Moving the tag allocation from the plug phase also removes the back pressure point in Falcon, and hence the IO issuing thread could potentially keep submitting as many requests as it can, and consume a lot of system resources such as memory.

To address this problem, Falcon proposes a per-drive limit. When the number of IOs increases beyond a high-pressure point, the thread stops IO submission in FML and sleeps. The thread will become active again when the number of requests drops below a low-pressure point in the whole IO stack, thus controlling the in-flight IO count in the whole IO pipeline. The number of requests in the bio-queue is used to determine the pressure point. For a multi-drive volume, the pressure point is equal to the product of per-drive pressure point and drive count.

The pressure point is a different threshold than the watermark. The former is about when to block IO processing thread, while the latter is used to decide when to do synchronous or parallel dispatch.

6 Experiments

The machine used for the experiments has dual-socket of Intel Xeon CPU E5-2620 2GHz with six cores each, thus total 24 threads due to hyper-threading, and 32GB DRAM. We use eight Samsung EVO 850 500GB SSDs connected using LSI SAS9300-8i HBA which supports SATA III SSDs. The system also has a Samsung 950 pro 512GB NVMe SSD with PCI 3.0 interface, two Intel AHCI SATA III ports, four SATA II ports, and four SCU ports which supports SATA II SSDs. We use four Intel 520 120GB SSDs for testing SCU ports.

We run the tests on Linux kernel version 4.4.0 with the Blk-mq block layer [1] (which performs better than the single-queue block layer). The blk-mq architecture has been completely integrated with SCSI layer and other drivers (called scsi-mq) in this kernel. Currently, blk-mq does not have any configurable IO scheduling policy.

We have implemented a prototype of Falcon in about 600 lines of C code with the aforementioned Linux kernel. We use the *md* software as the volume manager with

default stripe size of 4KB in a RAID-0 configuration. By default we use raw volumes, and also evaluate ext4 and XFS file system in a number of cases.

6.1 Microbenchmarks

We use a modified FIO in these tests. FIO [12] provides a number of IO engines such as AIO and synchronous POSIX IO and outputs a number of parameters including throughput, IOPS, and latency. However, FIO spends a lot of time in userspace (more than 35%), thus can not submit IOs as fast as a single application thread can otherwise. To address this problem, we modify FIO to instead simply replay the traces as fast as possible.

Ext4 File Throughput. The per-inode lock on ext4 File System does not allow Linux to saturate the 8-SSD volume even using multiple application IO threads. However, Falcon can saturate the volume using just one application IO thread, as shown in Figure 14. The improvement is due to parallelism at block layer tasks (sort, merge, tag allocation and dispatch). Overall, Falcon achieves 1.77× and 1.59× random read and write throughput compared to Linux on an ext4 file in 8-SSD volume.

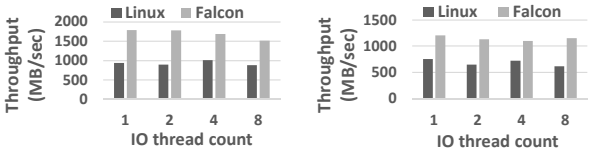


Figure 14: Random IO on an Ext4 file in 8-SSD volume

Buffered Write Throughput. Figure 15 shows improvement in buffered write throughput when 8 application IO threads are doing random write on multi-SSD volumes. Again, Linux is not able to achieve beyond 800 MB/s throughput on 8-SSD system because Linux buffer cache management allows only one *pdflush* thread to write the dirty buffer to the volume. In contrast, Falcon achieves 1.38× and 1.59× improvement compared to Linux in raw 4-SSD and 8-SSD volumes.

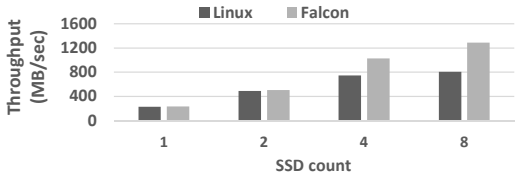


Figure 15: Buffered random write throughput scaling

Varying SSD Count. Figure 16 shows that Falcon delivers performance improvement by 1.92×, 3.65× 6.02× for random read, and 1.86×, 3.34× and 6.29× for random write in 2-SSD, 4-SSD and 8-SSD volumes over one SSD respectively. This clearly indicates that Falcon is scalable when more SSDs are added to the volume.

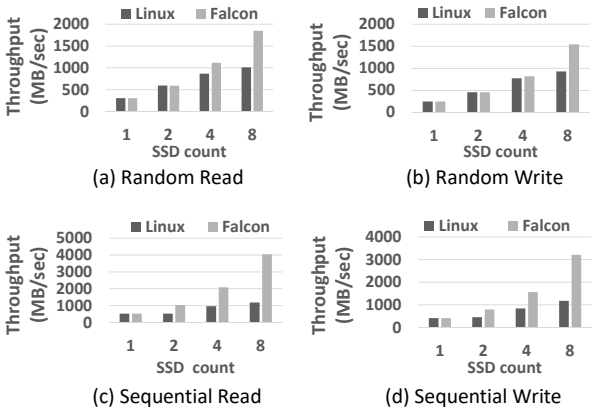


Figure 16: IO Scalability by varying the number of SSDs

For the 8-SSD volume (LSI HBA), Falcon achieves 1.83×, 1.66×, 3.42× and 2.73× speedup for random read, random write, sequential read and sequential write, respectively. When using the SCU controller for 4-SSD volume, Falcon can also achieve 1.25×, 1.08×, 1.59× and 1.85× respectively, again saturating the volume completely.

Varying Stripe Size. Figure 17 shows the random and sequential IO throughput on an 8-SSD volume for a variety of stripe size configurations. Random IO (4KB IO size) is highly susceptible to stripe size configuration as a better IO distribution to all the member drives will maximize the IO, while a skewed distribution would not. Figure 17 shows that 4KB stripe size is the best, while 32KB stripe is the worst for the random IO pattern generated by the FIO. In the best case, Falcon provides 1.83× and 1.66× random read and write throughput respectively.

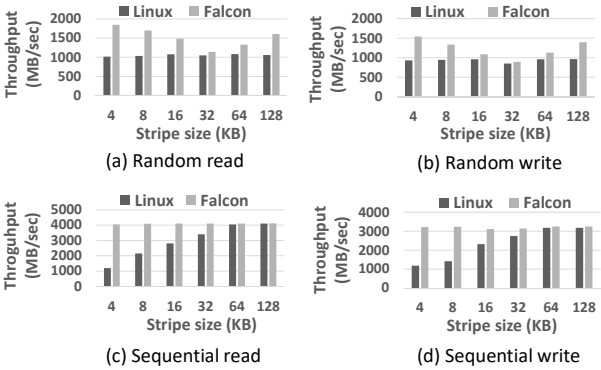


Figure 17: Throughput for various stripe sizes in 8-SSD volume

Sequential IO (64KB IO size) generates uniform IO load to each member drive of the volume. For 4KB stripes, Falcon provides 3.42× and 2.73× sequential read and write throughput respectively compared to Linux. Falcon saturates the 8-SSD volume irrespective of stripe size, while Linux can saturate the volume only when the IO size is smaller than or equal to the stripe size. This is because the Linux block layer is written with the assumption of a single drive. In other words, the

Linux block layer assumes that split has been performed for the IOs larger than 1 MB (maximum IO size in the block layer), and wrongly determines that further merging would no longer be needed on split IOs. In contrast, FBL enables the merge in case of IO split, and enjoys the performance benefit from the full sequential IOs.

Advantage and Scalability. There are two important observations when running Falcon on an 8-SSD volume. First, Falcon saturates the sequential read/write completely (Figure 16). Second, for random read and write tests, Falcon achieves 97.6% and 98.9% throughput of an ideal system, where one application IO thread is dedicated to submit batched IOs to each SSD independently, so that IO skewness is not the concern.

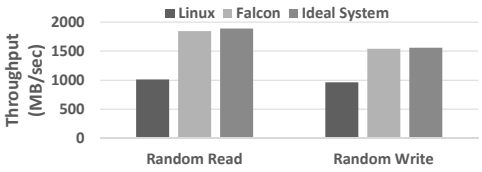


Figure 18: Random read comparsion of Linux and Falcon with an ideal system on an 8-SSD volume.

To understand the maximum random throughput achievable by an application IO thread, we run the same experiment in a null block device. The device is a standard Linux driver without any backup storage, and acknowledges the IOs as soon as received. Falcon can provide up to 3.7GB per second random read, which would be roughly equivalent to the aggregate throughput of 16 SSDs. However, it should be noted that null block device avoids many operations which otherwise were needed for a real volume. We expect that spawning more than one application IO thread will saturate 16 or more SSDs.

6.2 Application Performance

All the tests in this section are performed in XFS File System. For Linux, XFS tends to outperform ext4 for parallel reads as it does not acquire an inode lock. Since Falcon always deploys a single application IO thread, the choice of file system would not matter.

Utility Applications. We choose *copy* and *tar* to show the effectiveness of Falcon. Copy represents parallel data copying between the volume and a 24GB RAM disk. Specifically, *CopyTo* copies the 24GB file from the RAM disk to the volume. *CopyFrom* does the reverse. On the other hand, *Tar* and *Untar* use pbzip2, a parallel implementation of bzip2. As shown in Figure 19(a), Falcon speeds up CopyFrom, CopyTo, Tar and Untar by 1.63×, 2.81×, 1.29× and 1.09× respectively. The benefits are lower for Tar and Untar as they are more CPU intensive.

Filebench. We also run Fileserver (2:1 read/write ratio), Webserver (mostly read with random appends) and Webproxy (read only) personality in Filebench suite of

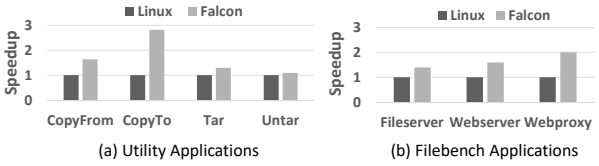


Figure 19: Application performance on 8-SSD volume

benchmarks. Figure 19(b) shows that Falcon performs 1.39×, 1.60× and 2× better than Linux when the benchmarks ran on around 64GB of data.

Graph Processing. We choose G-Store [20], a semi-external graph processing system as a representative use-case for high throughput application to demonstrate the effect of Falcon. In particular, we evaluate four different graph algorithms including breadth-first search (BFS) [7, 14], kCore [33, 29], connected component (CC) [35] and page rank (PR) [2] algorithms. BFS and kCore generate very high random IOs on graph data, while CC and PR require mostly sequential IOs. The experiments are on a undirected kronecker graph of scale 28 and edge factor 16 [14].

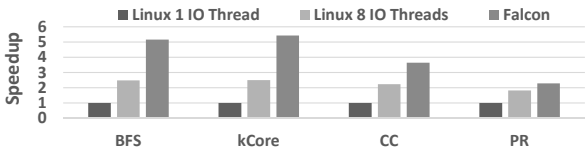


Figure 20: Graph Processing Performance

Figure 20 shows that Falcon significantly speeds up the graph processing by 4.12× and 1.78× compared to using one and eight IO threads in Linux. In particular, Falcon achieves more than 5× speedup for BFS and k-Core compared to using one IO thread in Linux, and 2× improvement over 8 IO threads. In the case of CC and PageRank, Falcon also provides 2 to 3× improvement over using a single IO thread.

6.3 Server IO Traces

We run application traces collected at University of Massachusetts Amherst [41] and Florida International University [19]. Table 2 provides the information on these traces. UM-Financial1 and UM-Financial2 represent the OLTP type applications, while UM-Websearch1 and UM-Websearch2 are websearch traces. On the other hand, FIU-Home, FIU-Mail, FIU-Webuser and FIU-Web-vm represent the traces from home directory, mail, web user, and webmail proxy and online course management system. The traces contain the lower-level IOs, i.e., at the logical block address. Almost all the write operations are caused by *kjournald* or *pdflush* daemons which run in the kernel space. These daemons behave more like a batched Linux AIO interface.

We replay the traces by submitting IOs as fast as the system allows. Figure 21 shows *Falcon* can extract

Trace Name	Read (%)	IO size range	Size (GB)
UM-Financial1	23.16	512B - 16715KB	17.22
UM-Financial2	82.34	512B - 256.5KB	8.44
UM-Websearch1	99.98	512B - 1111KB	15.24
UM-Websearch2	99.98	8KB - 32KB	65.82
FIU-Home	1.00	512B - 512KB	34.58
FIU-Mail	8.58	4KB - 4KB	86.64
FIU-Webuser	10.33	4KB - 128KB	30.94
FIU-Web-vm	21.8	4KB - 4KB	54.52

Table 2: IO traces summary

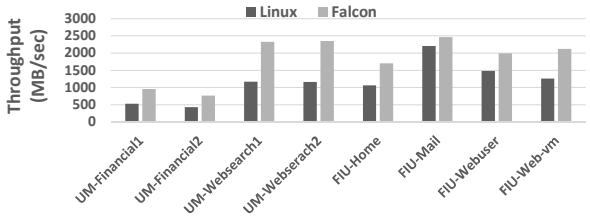


Figure 21: Trace replay throughput on 8-SSD volume

more throughput for all IO traces. Overall Falcon performs 1.67× better over Linux. UM-Financial1, UM-Financial2 has small throughput as they do many 512 byte random IOs. FIU-Home traces are almost write-only and random IO. Others are mix of random and sequential IOs.

7 Related Work

Prior works [4, 5, 46, 37, 43, 45, 48, 17, 36, 47] mostly aim to improve the IO stack for one drive by proposing changes in the IO stack and/or hardware, leaving behind a number of issues pertaining to multi-SSD volumes. Different from these approaches, we focus on multi-SSD volumes and aim to achieve both the IO scalability and the ease of programming.

Application-managed IO approach [49, 50, 23] partitions a file in different SSDs and proposes a userspace abstraction to aggregate the file content. As a result, it introduces a lot of complexity in the application, and lacks support of POSIX file system [49]. Also, there are application level restrictions, such as only integer number of processing cores for each SSD (e.g., one compute thread for each SSD in Graphene [23]).

Various works [40, 27, 22, 34] have identified the importance of IO stack optimization, and have proposed various changes including the block layer to accelerate application performance. A nice description of the time spent on different layers of the IO platform has been analyzed in [13]. Problems with sub-page buffered write is identified [3] and techniques have been proposed to improve the performance. Wang et al. [44] propose fairness and efficiency in tiered storage system. Our work is in-tune with these efforts and especially identifies improvements in the IO stack for multi-SSD volumes.

Linux kernel developers have made many improvement in various locking semantics [15, 16, 24], however, locking in parallel IO from the same file still remains an issue as observed by Min et al. [28] in a many-core system. An enhanced storage layer has been proposed in [8] that exports information to file systems to bridge the information gap. Our system utilizes such information within a new layer (FML) to improve the performance in multi-SSD volumes.

Storage area network (SAN) solutions provide aggregated SSDs as block device services, and scale in terms of multiple clients. However, the local file system at client side will still have all the constraints that we have discussed. Chen et al. [6] have developed new batched IO interface, and integrated it with NFS compound procedure using a userspace NFS client. In such cases, Falcon will become a natural choice of IO stack to provide a better client-side IO stack to take advantage of the faster SAN/NFS storage.

Big data applications such as WiscKey (a key-value store) [25] deploy a complex mechanism of a thread pool to serve IOs in one SSD. To scale to multiple SSDs, they need to fine-tune the number of threads. Falcon will enable such applications to move towards just one thread, and can saturate more SSDs if AIO interfaces are used, thereby providing simplicity and scalability both.

Lastly, many graph applications [21, 32] bypass the random IO problem in multi-SSD volumes by fetching the whole data in each iteration. However, Vora et al. [42] show that the performance of many graph algorithms can be improved by doing selective random IO. We believe that Falcon will become a platform of choice for many IO-intensive applications including graph analytics.

8 Conclusion

In this work, we have identified that the separation of two IO processing tasks, i.e., IO batching and IO serving in the block layer, holds the key to improve the throughput in multi-SSD volumes. To achieve this goal, *Falcon* proposes a new IO stack to enforce per-drive processing that improves the IO stack performance and parallelizes the IO serving tasks. Compared to current practice, Falcon significantly accelerates a variety of applications from utility applications to graph processing, and also shows strong scalability across different numbers of drives, and various storage controllers.

9 Acknowledgments

The authors thank the USENIX ATC'17 reviewers and our shepherd Sorav Bansal for their suggestions. This work was supported in part by National Science Foundation CAREER award 1350766 and grant 1618706.

References

[1] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet. Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 22:1–22:10, New York, NY, USA, 2013. ACM.

[2] S. Brin and L. Page. The Anatomy of a Large-scale Hypertextual Web Search Engine. In *Proceedings of the Seventh International Conference on World Wide Web 7*, Amsterdam, The Netherlands, 1998.

[3] D. Campello, H. Lopez, R. Koller, R. Rangaswami, and L. Useche. Non-blocking Writes to Files. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 151–165, Santa Clara, CA, Feb. 2015. USENIX Association.

[4] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. 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*, MICRO '43, pages 385–395, Washington, DC, USA, 2010. IEEE Computer Society.

[5] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson. Providing Safe, User Space Access to Fast, Solid State Disks. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 387–400, New York, NY, USA, 2012. ACM.

[6] M. Chen, D. Hildebrand, H. Nelson, J. Saluja, A. S. H. Subramony, and E. Zadok. vNFS: Maximizing NFS Performance with Compounds and Vectorized I/O. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 301–314, Santa Clara, CA, 2017. USENIX Association.

[7] 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*.

[8] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 177–190, Berkeley, CA, USA, 2002. USENIX Association.

[9] Disk Controller Queue-Depth. `http://www.yellow-bricks.com/2014/04/17/disk-controller-features-and-queue-depth/`, 2016.

[10] EXT4 File System. `https://ext4.wiki.kernel.org/index.php/Main_Page`, 2016.

[11] Facebook RocksDB. `http://rocksdb.org/`, 2016.

[12] Flexible I/O Tester. `https://github.com/axboe/fio`, 2016.

[13] A. P. Foong, B. Veal, and F. T. Hady. Towards SSD-Ready Enterprise Platforms. In *VLDB*, 2010.

[14] Graph500. `http://www.graph500.org/`.

[15] C. Jonathan. JLS: Increasing VFS Scalability. `https://lwn.net/Articles/360199/`. 2009.

[16] C. Jonathan. Dcache scalability and RCU-walk. `https://lwn.net/Articles/419811/`. 2010.

[17] J. Kang, B. Zhang, T. Wo, W. Yu, L. Du, S. Ma, and J. Huai. SpanFS: A Scalable File System on Fast Storage Devices. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 249–261, Santa Clara, CA, 2015. USENIX Association.

[18] H.-J. Kim, Y.-S. Lee, and J.-S. Kim. NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO, June 2016. USENIX Association.

[19] R. Koller and R. Rangaswami. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. In *8th USENIX Conference on File and Storage Technologies (FAST 10)*. USENIX Association, Feb. 2010.

[20] P. Kumar and H. H. Huang. G-Store: High-Performance Graph Store for Trillion-Edge Processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[21] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-Scale Graph Computation on Just a PC. In *OSDI*, 2012.

[22] W. Li, G. Jean-Baptise, J. Riveros, G. Narasimhan, T. Zhang, and M. Zhao. CacheDedup: In-line Deduplication for Flash Caching. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 301–314, Santa Clara, CA, Feb. 2016. USENIX Association.

[23] H. Liu and H. H. Huang. Graphene: Fine-Grained IO Management for Graph Computing. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, 2017.

[24] W. Long. [PATCH] dcache: Translating dentry into pathname without taking rename_lock. `https://lkml.org/lkml/2013/9/4/471`. 2013.

[25] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, Santa Clara, CA, Feb. 2016. USENIX Association.

[26] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, 2017.

[27] S. Mandal, G. Kuenning, D. Ok, V. Shastry, P. Shilane, S. Zhen, V. Tarasov, and E. Zadok. Using Hints to Improve Inline Block-layer Deduplication. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 315–322, Santa Clara, CA, Feb. 2016. USENIX Association.

[28] C. Min, S. Kashyap, S. Maass, and T. Kim. Understanding Many-core Scalability of File Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 71–85, Denver, CO, 2016. USENIX Association.

[29] A. Montresor, F. De Pellegrini, and D. Miorandi. Distributed k-Core Decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 2013.

[30] J. Oh, W.-S. Han, H. Yu, and X. Jiang. Fast and Robust Parallel SGD Matrix Factorization. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 865–874. ACM, 2015.

[31] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[32] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric Graph Processing using Streaming Partitions. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), 2013*.

[33] A. E. Sariyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek. Streaming algorithms for k-core decomposition. *Proceedings of the VLDB Endowment*, 6(6):433–444, 2013.

[34] E. Seppanen, M. T. O'Keefe, and D. J. Lilja. High Performance Solid State Storage Under Linux. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, May 2010.

[35] Y. Shiloach and U. Vishkin. An O(log n) Parallel Connectivity Algorithm. *Journal of Algorithms*, 1982.

[36] D. I. Shin, Y. J. Yu, H. S. Kim, J. W. Choi, D. Y. Jung, and H. Y. Yeom. Dynamic Interval Polling and Pipelined Post I/O Processing for Low-Latency Storage Class Memory. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems*, San Jose, CA, 2013. USENIX.

[37] W. Shin, Q. Chen, M. Oh, H. Eom, and H. Y. Yeom. OS I/O Path Optimizations for Flash Solid-state Drives. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 483–488, Berkeley, CA, USA, 2014. USENIX Association.

[38] SNIA. `http://nvmexpress.org/wp-content/uploads/2013/04/NVM_whitepaper.pdf`,.

[39] Specification for Supermicro Product(X9DRG-HF). `http://www.supermicro.com/products/motherboard/Xeon/C600/X9DRG-HF.cfm`, 2016.

[40] I. Stefanovici, B. Schroeder, G. O'Shea, and E. Thereska. sRoute: Treating the Storage Stack Like a Network. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 197–212, Berkeley, CA, USA, 2016. USENIX Association.

[41] UMass Trace Repository. `http://traces.cs.umass.edu/index.php/Storage/Storage`, 2012.

[42] K. Vora, G. Xu, and R. Gupta. Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 507–522, Denver, CO, 2016. USENIX Association.

[43] D. Vučinić, Q. Wang, C. Guyot, R. Mateescu, F. Blagojević, L. Franca-Neto, D. L. Moal, T. Bunker, J. Xu, S. Swanson, and Z. Bandić. 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 (FAST 14)*, pages 309–315, Santa Clara, CA, 2014. USENIX.

[44] H. Wang and P. Varman. Balancing Fairness and Efficiency in Tiered Storage Systems with Bottleneck-Aware Allocation. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 229–242, Santa Clara, CA, 2014. USENIX.

[45] J. Yang, D. B. Minturn, and F. Hady. When Poll is Better Than Interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association.

[46] Y. J. Yu, D. I. Shin, W. Shin, N. Y. Song, J. W. Choi, H. S. Kim, H. Eom, and H. Y. Yeom. Optimizing the Block I/O Subsystem for Fast Storage Devices. *ACM Transaction on Computer System (TOCS)*, 32(2):6:1–6:48, June 2014.

[47] Y. J. Yu, D. I. Shin, W. Shin, N. Y. Song, H. Eom, and H. Y. Yeom. Exploiting Peak Device Throughput from Random Access Workload. In *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'12, pages 7–7, Berkeley, CA, USA, 2012. USENIX Association.

[48] J. Zhang, J. Shu, and Y. Lu. ParaFS: A Log-Structured File System to Exploit the Internal Parallelism of Flash Devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 87–100, Denver, CO, 2016. USENIX Association.

[49] D. Zheng, R. Burns, and A. S. Szalay. Toward Millions of File System IOPS on Low-cost, Commodity Hardware. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 69:1–69:12, New York, NY, USA, 2013. ACM.

[50] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[51] X. Zhu, W. Han, and W. Chen. GridGraph: Large-scale Graph Processing on a Single Machine Using 2-level Hierarchical Partitioning. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference*, 2015.

deTector: a Topology-aware Monitoring System for Data Center Networks

Yanghua Peng
The University of Hong Kong

Ji Yang
Xi'an Jiaotong University

Chuan Wu
The University of Hong Kong

Chuanxiong Guo
Microsoft Research

Chengchen Hu
Xi'an Jiaotong University

Zongpeng Li
University of Calgary

Abstract

Troubleshooting network performance issues is a challenging task especially in large-scale data center networks. This paper presents *deTector*, a network monitoring system that is able to detect and localize network failures (manifested mainly by packet losses) accurately in near real time while minimizing the monitoring overhead. *deTector* achieves this goal by tightly coupling detection and localization and carefully selecting probe paths so that packet losses can be localized only according to end-to-end observations without the help of additional tools (*e.g.*, tracert). In particular, we quantify the desirable properties of the matrix of probe paths, *i.e.*, coverage and identifiability, and leverage an efficient greedy algorithm with a good approximation ratio and fast speed to select probe paths. We also propose a loss localization method according to loss patterns in a data center network. Our algorithm analysis, experimental evaluation on a Fattree testbed and supplementary large-scale simulation validate the scalability, feasibility and effectiveness of *deTector*.

1 Introduction

A variety of services are hosted in large-scale data centers today, *e.g.*, search engines, social networks and file sharing. To support these services with high quality, data center networks (DCNs) are carefully designed to efficiently connect thousands of network devices together, *e.g.*, a 64-ary Fattree [9] DCN has more than 60,000 servers and 5,000 switches. However, due to the large network scale, frequent upgrades and management complexity, failures in DCNs are the norm rather than the exception [21], such as routing misconfigurations, link flaps, etc. Among these failures, those leading to user-perceived performance issues (*e.g.*, packet losses, latency spikes) are among the first priority to be detected and eliminated promptly [27, 26, 21], in order to maintain high quality of service (QoS) for users (*e.g.*, no more than a few minutes of downtime per month [21]) and to increase revenue for operators.

Rapid failure recovery is not possible without a good network monitoring system. There have been a number of systems proposed in the past few years [36, 26, 37, 48]. Several limitations still exist in these systems that prohibit fast failure detection and localization.

First, existing monitoring systems may fail to detect one type of failures or another. Traditional passive monitoring approaches, such as querying the device counter via SNMP or retrieving information via device CLI when users have perceived some issues, can detect clean failures such as link down, line card malfunctions. However, gray failures may occur, *i.e.*, faults not detected or ignored by the device, or malfunctioning not properly reported by the device due to some bugs [37]. Active monitoring systems (*e.g.*, Pingmesh [26], NetNORAD [37]) can detect such failures by sending end-to-end probes, but they may fail to capture failures that cause low rate losses, due to ECMP in data center (§2).

Second, probe systems such as Pingmesh and NetNORAD inject probes between each pair of servers without selection, which may introduce too much bandwidth overhead. In addition, they typically treat the whole DCN as a black box, and hence require many probes to cover all parallel paths between any server pair with high probability.

Third, failures in the network can be reported in these active monitoring systems, but the exact failure locations cannot be pinpointed automatically. The network operator typically learns a suspected source-destination server pair once packet loss happens. Then she/he needs to resort to additional tools such as tracert to verify the issue and locate the faulty spot. However, it may be difficult to play back the issues due to transient failures. Hence this diagnosis approach (*i.e.*, separation of detection and localization) may take several hours or even days to pinpoint the fault spot [21], yet ideally the failures should

be repaired as fast as possible before users complain.

A desirable monitoring system in a DCN should meet three objectives: exhaustive failure detection (*i.e.*, detecting as many types of losses as possible), low overhead and real-time failure localization. In this paper, we seek to investigate the following question: if we are aware of the network topology of a DCN, can we design a much better network monitoring system that achieves all these goals? Our answer is *deTector*, a topology-aware network monitoring system that we design, implement and evaluate following the three design objectives. The secret weapon of *deTector* is a carefully designed probe matrix (§4), which achieves good link coverage, identifiability and evenness. *deTector* is designed to detect and localize network failures manifested by user-perceptible performance problems such as packet losses and latency spikes in large-scale data centers. We mainly focus on packet loss in this paper, but *deTector* can also handle latency issues by treating a round trip time (RTT) larger than a threshold as a packet loss. Throughout the paper, we use "failure localization", "fault localization" and "loss localization" interchangeably. Specifically, we make the following contributions in developing *deTector*.

▷ As compared to the existing active monitoring systems adopting end-to-end probes (*e.g.*, Pingmesh [26], NetNORAD [37]), we treat each switch instead of the whole network as a blackbox, *i.e.*, our system requires the knowledge of the network topology and routing protocols in a DCN (*i.e.*, topology-aware) and we use source routing to control the probing path. In order to achieve real-time failure localization, we couple detection and localization closely and only rely on end-to-end measurements to localize failures without the help of other tools (*e.g.*, fbtracert [3]). To make it possible, we quantify several desirable properties of probe matrix (*e.g.*, identifiability) and propose a greedy algorithm to minimize probe cost. To address the scalability issue in DCNs, we apply several optimization heuristics and exploit characteristics of the DCN topology to accelerate path computation (§4).

▷ We modify a failure localization algorithm based on packet loss characteristics in large-scale data centers. Compared to the existing algorithms, our algorithm runs within seconds and achieves higher accuracy and lower false positive rate (§5).

▷ We implement and evaluate our system on a 4-ary Fattree testbed built with 20 switches. The experiments show that *deTector* is practically deployable and can accurately localize failures in near real time with less probe overhead, *e.g.*, for 98% accuracy, *deTector* requires 3.9x and 1.9x times fewer probes than Pingmesh and NetNORAD while localizing failures 30 seconds in advance without the use of other loss localization tools. Our supplementary simulation further shows that *deTector* achieves greater than 98% accuracy in failure localization with a less than 1% false positive ratio for most failures in large-scale DCNs (§6). We have open sourced *deTector* [6].

2 Motivation

DCNs are usually multi-stage Clos networks with multiple paths between commodity servers for load balancing and fault tolerance [9, 22, 26, 45]. Each DCN has its favorable routing protocols for path selection. For example, in a Fattree topology [9] and a VL2 topology [22], the shortest paths between any two ToRs are typically used in practice [30]. We describe how existing monitoring systems fall short in achieving the three design objectives. Table 1 shows detailed comparison among *deTector* and the existing systems.

The passive approach stores packet statistics on switch counters, which are polled from SNMP or CLI periodically. In Fig. 1, if link *AB* is down, the switch counters will show a lot of packet losses. However, if the failure is a gray failure rather than link down, it may go undetected. For example, when silent packet drops occur, the switch do not show any packet drop hints (*e.g.*, syslog errors) due to various reasons (*e.g.*, ASIC deficit), and hence SNMP data may not be fully trustworthy [26]. Furthermore, switches counters can be noisy, such that problems identified by this approach may or may not lead to end-to-end delay or loss perceived by users.

Pingmesh and NetNORAD adopt an end-to-end probing approach to measure network latency and packet loss. Pingmesh selects probe paths by constructing two complete graphs within a DCN: one includes all servers under the same ToR switch (i.e., the switch in the edge layer in Fig. 1) and the other spans all ToR switches. NetNORAD is similar to Pingmesh but places pingers in a few pods instead of all servers. Their approaches simplify the design but bring quite significant overhead (§6). Although gray failures can be captured, it is difficult to detect failures causing low rate losses (*e.g.*, 1%) of a link, when ECMP is adopted in the DCN: there are many paths be-

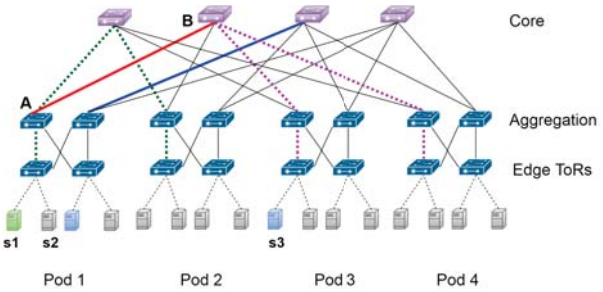


Figure 1: A 4-radix Fattree topology: failure on link *AB* can be detected by sending probes from $s1$ to $s3$.

Table 1: Comparison among *deTector* and existing representative monitoring systems

	Gray failures	Low rate loss	Failure localization	Transient failures	Timeliness	Overhead
SNMP/CLI	No	No	Yes	Yes	minutes	switch resources
Pingmesh [26]	Yes	No	No, need Netbouncer	No	minutes	many probes
NetNORAD[3]	Yes	No	No, need fbtracert	No	minutes	many probes, switch CPU
deTector	Yes	Yes	Yes	Yes	near real-time	minimal probes

tween a pair of servers, low-rate losses on a particular link may not affect much the overall end-to-end loss rate between the two servers.

The exact location of losses cannot be pinpointed using Pingmesh or NetNORAD, since they do not know which paths the probes take (*e.g.*, due to ECMP). Therefore, other tools such as Netbouncer [4] and fbtracert [3] are needed, which send additional probes to play back the losses. These post-alarm tools may fail to pinpoint transient failures, those caused by transient bit errors, non-atomic rule updates or network upgrade (*e.g.*, a transient inconsistency between the link configuration and routing information [21]). To pinpoint such failures, close coupling of detection and localization is required, so that losses are localized only according to detection data, instead of additional probes after detection alarms. Such coupling further enables near real-time fault localization.

3 System Design

3.1 Architecture

deTector includes four loosely coupled components: a controller, a diagnoser, pingers and responders, as depicted in Fig. 2.

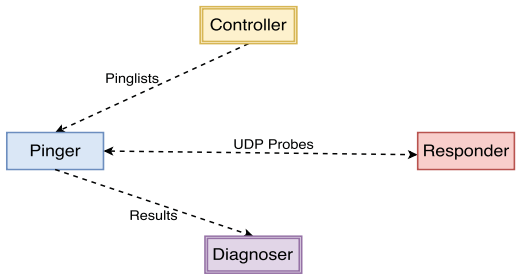


Figure 2: System architecture

Controller. The logical controller periodically constructs the probe matrix indicating the paths for sending probes (see §4 for details). We mainly focus on failure localization on links inter-connecting switches, as the fault on a link connecting a server with a ToR switch can be easily identified as discussed in the next paragraph. The probe matrix indicates paths between ToRs. Since we do not rely on ToRs with ping capability, probes are sent by 2–4 selected servers (pingers) under each ToR.

Pinger. Each pinger receives the pinglist from the controller, which contains server targets, probe format and ping configuration (§6.1). The probe paths from a ToR switch to different destinations are distributed among pinglists of pingers under the ToR switch, with each path distributed to at least 2 pingers for fault tolerance. In this way, in case that one pinger is down, other pingers in the same rack can still probe the paths, avoiding any large drop in link coverage. To detect failure on links connecting servers and the respective ToRs, pingers are also responsible for probing other servers under the same ToR. The number of probe paths for each pinger is no more than a hundred even for a large DCN (§4.4). The probe packets are sent over UDP. Though TCP is used to carry most traffic in a DCN, the DCN does not differentiate TCP and UDP traffic (*e.g.*, the forwarding behavior) in the vast majority of cases [37, 26], and hence UDP probes can also manifest network performance. When a pinger detects a probe loss, it confirms the loss pattern by sending two probe packets of the same content additionally.

Responder. The responder is a lightweight module running on all servers. Upon receiving a probe packet, the responder echoes it back. A responder does not retain any states and all probing results are logged by pingers.

Diagnoser. Each pinger records packet loss information and sends it to the diagnoser for loss localization. These logs are saved into a database for real-time analysis and later queries. The diagnoser runs the PLL algorithm (§5) to pinpoint packet losses and estimates the loss rates of suspected links.

For the controller and the diagnoser to be fault-tolerant and scalable, we can use existing solutions (*e.g.*, Software Load-Balancer [41, 26]).

3.2 Workflow Overview

deTector works in three steps in cycles: path computation, network probing and loss localization.

Path computation. At the beginning of each cycle, the controller reads the data center topology and server health from data center management service (*e.g.*, [31]), and selects the minimal number of probe paths (§4). The controller then selects pingers in each ToR, constructs and dispatches the pinglists to them.

Network probing. Next, probe packets are sent along the specified paths across the DCN. Since data center usually adopts ECMP for load balancing, we have to use source routing to control the path traveled by each probe packet, which can be implemented using various

methods.[1] A general and feasible solution is to employ packet encapsulation and decapsulation to create end-to-end tunnels, though it may cause encapsulating packets twice in virtualized networks created by VXLAN [1] or NVGRE [2]. Take the Fattree network in Fig. 1 as an example: fixing a core switch, there is only one path between two inter-pod servers; we can use IP-in-IP encapsulation to wrap the probe on a server; after the packet arrives at the core switch, the outer header is removed and the packet is routed to the real destination. Such a source routing mechanism incurs little overhead on servers and core switches.

Loss localization. The probe loss measurements are aggregated and analyzed by our loss localization algorithm (§5) on the diagnoser. We pinpoint the faulty links, estimate the loss rates, and send alerts to the network operator for further action (e.g., examining switch logs).

4 Probe Matrix Design

The main limitation of existing monitoring systems is that the probe path selection is far from optimum, such that not enough useful information can be collected and additional probes are needed to reproduce losses for localization. In this section, we elaborate how we carefully select probe paths to overcome such a limitation.

4.1 Problem

Consider a data center network graph $G = (V, E)$, where V is the set of switches and E is the set of links. $\mathbf{R}$ is the $m \times n$ routing matrix defined by

$$\mathbf{R}_{i,j} = \begin{cases} 1 & \text{if link } j \text{ is on path } i \\ 0 & \text{otherwise} \end{cases}$$

where m is the number of paths and $n = |E|$ is the number of links. The possible paths and the routing matrix are decided by the routing protocols employed in the data center, *e.g.*, ECMP is typically used to exploit $k^2/4$ parallel paths between any two ToRs in a k-ary Fattree. Fig. 3 gives a routing matrix $\mathbf{R}$ with 3 paths and 3 links. Note that each link in a DCN is typically bi-directional. Once we select a path from server $s1$ to server $s2$ and send a probe, the reverse path from $s2$ to $s1$ is automatically selected, since the response packet can probe faults along the reverse direction. When we identify that link AB has failed, it implies that the failure may lie in either direction of the link, switch A, or switch B.

[1] Source routing protocols have been designed in some DCNs like BCube [24] and DCell [25]; [30, 32] introduce other solutions for explicit path control.

$$\mathbf{R} = \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \end{array} \begin{array}{c} \begin{array}{ccc} l_1 & l_2 & l_3 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \end{array} \rightarrow \mathbf{R'} = \begin{array}{c} \begin{array}{ccc|ccc} l_1 & l_2 & l_3 & l_{12} & l_{13} & l_{23} \end{array} \\ \left(\begin{array}{ccc|ccc} 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{array}\right) \end{array}$$

Figure 3: Extend routing matrix with virtual links

Problem 1 *Given a DCN routing matrix* $\boldsymbol{R}$*, select a set of paths to construct a probe matrix* $\boldsymbol{P}$*, such that* $\boldsymbol{P}$ *simultaneously (1) minimizes the number of paths, and achieves (2)* α*-coverage and (3)* β*-identifiability.*

Minimizing the number of probe paths is desirable for minimizing network bandwidth consumption and analysis overhead, such that we may finish probing and diagnosing the entire DCN in merely a few minutes. Under the same probing bandwidth budget, it allows each pinger to probe the same set of paths more frequently.

α*-coverage* requires that each link is covered by at least α paths in the probe matrix. Covering a link multiple times brings higher statistical accuracy for loss detection, as well as better resilience to pinger failures (since a link is more likely to be covered by probes from multiple pingers).

β*-identifiability* states that the simultaneous failures of any (no more than) β links in the DCN can be localized correctly. For the routing matrix in Fig. 3, suppose we select p_1 and p_2 to constitute the probe matrix, *i.e.*, the probe matrix contains the first two rows of $\mathbf{R}$. If 2 or more links fail simultaneously, the faulty links cannot be correctly identified, as the observation from the end is the same, *i.e.*, packet losses are observed on both paths. On the other hand, if only one link is faulty, the bad link can be identified effectively: losses are observed on both paths, p_1, or p_2 if link 1, 2, or 3 is faulty, respectively. Therefore, the probe matrix achieves 1-identifiability, but not 2 or higher identifiability. Better identifiability contributes to higher accuracy of loss localization.

We find that Problem 1 is NP-hard for general DCNs as the Minimum Set Cover Problem is a special case of the problem. We hence resort to an approximation algorithm to compute the probing path, which is at the heart of *deTector*.

4.2 PMC Algorithm

We extend a well-known greedy algorithm [13] for constructing a probe matrix achieving 1-identifiability to one achieving β-identifiability, as well as α-coverage using a minimal number of probe paths.

In a probe matrix, a link belongs to a set of paths. To achieve 1-identifiability, the path sets of different links should all be different, so that losses can be observed on a particular set of paths to identify the faulty link. Recall that the set of links in our DCN is E. Once we select a path from the set of all feasible paths decided by the

routing matrix based on some criterion, it splits E into two subsets E_1 and E_2, containing links on the selected path and the other links, respectively. If we do not observe any packet loss on this path, it implies that all links in E_1 are good; otherwise, there must be at least one bad link in E_1. Similarly, we select another path to further split E_1 and E_2 into smaller subsets, and repeat this procedure. Eventually if we can obtain subsets each containing only one link, then the probe matrix constructed using the selected paths achieves 1-identifiability (since the set of paths traversing each link is unique); otherwise, there does not exist a 1-identifiable probe matrix in the DCN. Throughout the process, if we always select a path whose links are present in the largest number of link sets to further split the link sets as much as possible, we will end up with the minimal number of paths needed.

To achieve β-identifiability, we expand the DCN graph G with "virtual links". A virtual link is a combination of multiple physical links, and the set of paths a virtual link belongs to can be computed by "OR"-ing together the paths including the individual links [13]. For the example in Fig. 3, the original routing matrix **R** is extended to **R'** with three additional virtual links l_{12}, l_{13} and l_{23} added; the column corresponding to the virtual link l_{12} can be computed by "OR"-ing the two columns corresponding to links l_1 and l_2. For β-identifiability, $\sum_{2\le i\le\beta} C(|E|,i)$ virtual links should be added in the DCN graph (routing matrix), corresponding to all combinations of 2 to β links in the original graph. Then we can run the above algorithm for constructing 1-identifiable matrix based on the new routing matrix, and the resulting probe matrix achieves β-identifiability.

The probe matrix does not achieve even path coverage among the links yet. For example, for a 1-identifiable probe matrix constructed on a 64-ary Fattree, the gap between the maximal and minimal numbers of probing paths passing through any two links can be as large as 188. To achieve better evenness (*i.e.*, spreading paths and thus probe overhead evenly among the physical links), we introduce a link weight $w[link]$, denoting the number of paths that the link resides on, and ensure that it is no smaller than α for any physical link. We also define a score for each (extended) path, *i.e.*, the path includes virtual links from the extended routing matrix **R'**:

$$score(path) = \sum_{link\in path} w[link] - \#\text{ of link sets on } path \tag{1}$$

Here the link sets are the split link sets produced by the procedure above. We say that a link set is on a path if the link set contains at least one (physical or virtual) link of the path. Thus, a lower score indicates that the links on the path are not covered much by paths already selected and/or more link sets can be split if the path is selected in the above procedure. We strive to achieve better evenness among the links while guaranteeing α-coverage, by always selecting a path with the lowest score.

Algorithm 1 Probe Matrix Construction (PMC) Algorithm

Require: **R**, α, β

1: Initialize w, $score$ to **0**, $setnum$ to 1, $selpaths$ to $\emptyset$
2: **R'** $\leftarrow LINKOR(\mathbf{R},\beta)$
3: $paths \leftarrow$ all paths in **R'**, $physlinks \leftarrow E$
4: **while** $(setnum \neq |E| \parallel physlinks \neq \emptyset)$ && $paths \neq \emptyset$ **do**
5: **for** $path \in paths$ **do**
6: update $score[path]$ according to (1)
7: $path \leftarrow \text{argmin}_{path'\in paths} score[path']$
8: $selpaths \leftarrow selpaths \cup \{path\}$
9: $paths \leftarrow paths/\{path\}$
10: **for** $physlink$ on $path$ **do**
11: $w[physlink] \leftarrow w[physlink]+1$
12: **if** $w[physlink] > \alpha$ **then**
13: $physlinks \leftarrow physlinks/\{physlink\}$
14: update $setnum$ as the total number of link sets after split by $path$
15: return probe matrix constructed by paths in $selpaths$ (retaining only physical links on the paths)

Our Probe Matrix Construction algorithm, PMC, is summarized in Alg. 1. We first reduce the problem of constructing a β-identifiable matrix to one constructing 1-identifiable matrix, by adding virtual links to the original routing matrix of the DCN graph (line 2, where *LINKOR* denotes the method for extending routing matrix discussed above). Then in each iteration we update the score of each (extended) path (lines 5-6) and select a path which has the *minimal* score among all candidate paths (lines 7-8). We remove the selected path from the candidate path set (line 9), and update the weight of physical links ($w[physlink]$) on the selected path (lines 10-11) and the total number of link sets that the already selected paths can split into (line 14, which corresponds to the procedure discussed in the second paragraph of this subsection). If the number of paths that cover one (physical) link exceeds α, we remove the link from the set of all links (line 12-13). The loop stops when the probe matrix achieves α-coverage (i.e., the set *physlinks* is empty) and β-identifiability (i.e., the number of link sets split equals the number of links), or there are no more candidate paths (*i.e.*, the set *paths* is empty).

Theorem 1 *The PMC algorithm achieves* $(1-\frac{1}{e})$ *approximation of the optimum in terms of the total number of probe paths selected, where e is natural logarithm.*

We can prove Theorem 1 by showing that the score of a path set is monotone, submodular and non-negative.

The detailed proof is in the technical report [7]. In practice, the PMC algorithm performs much better than the $(1-\frac{1}{e}) \approx 0.63$ approximation ratio (§4.4). The issue of this algorithm, however, is the computation time. The time complexity of the algorithm is $O(m^2)$, where m is the number of paths, since in the worst case we may update the scores of all paths in each iteration and end up with selecting all paths. In a 64-radix Fattree, there are about 4.3×10^9 desirable paths among ToRs. As we will see in §4.4, the algorithm is still too slow for any data center at a reasonable scale, and we adopt a number of optimizations to further speed it up.

4.3 Algorithm Speedup

To speed up the PMC algorithm, we apply several optimizations based on the following three observations.

Observation 1 *Problem 1 can be divided into a series of subproblems.*

We can construct a bipartite graph according to the routing matrix: one partition corresponds to paths and the other consists of links; an edge exists between a path node and a link node if the link is on the path. We observe that if the routing matrix can be partitioned into sets of paths with no links in common, then the problem can be divided into independent subproblems. For example, in Fig. 1, paths traversing the red link have no link overlapping with paths traversing the blue link. Therefore, the bipartite graph can typically be divided into connected subgraphs and each subgraph represents a smaller routing matrix and hence a subproblem. Finding connected subgraphs can be done in linear time by traversing the bipartite graph once. Then the PMC algorithm can be applied to the subproblems in parallel.

Observation 2 *The score of each path is non-decreasing over all iterations.*

It can be proved that the score of a path is non-decreasing (Appendix A in [7]). Inspired by the CELF algorithm for outbreak detection in networks [38], we adopt a strategy called *lazy update* which defers the update of a path score as much as possible even though we know the score is outdated. Specifically, we maintain a min-heap for all paths with scores as the keys and only update the score of a path when the path is at the top of the heap. After score update, if the path still stays at the top of the heap, *i.e.*, the path has the minimal score among all available paths, we will select the path as a probe path, even though some path scores have yet to be updated. The correctness of this heuristic is guaranteed by submodularity of the score of a path set: the marginal gain provided by a path selected in the current iteration can not be larger than that provided by the path in the previous iteration.

Observation 3 *The DCN topology is typically symmetric.*

Due to symmetry, when a path is selected, all its topologically isomorphic paths can be selected. For example in Fig. 1, if the dashed green path spanning Pod 1 and Pod 2 is selected, then the dashed purple path spanning Pod 3 and Pod 4 may be a good choice too. This helps us reduce the scale of the problem since the routing matrix $\mathbf{R}$ can be reduced to a smaller matrix by excluding paths that are topologically isomorphic to other paths. For example, if the green path is in the matrix, we do not need to include the purple path. For this purpose, we first need to compute the symmetric components in a DCN graph. There are many fast algorithms available for symmetry discovery [17, 15], *e.g.*, O_2 [15] can finish computation within 5 seconds for a Fattree(100) DCN, and we only need to precompute it once for a DCN.

4.4 Performance

We run our PMC algorithm on a Dell PowerEdge R430 rack server with 10 Intel Xeon E5-2650 CPUs and 48GB memory, to test its running time and number of paths selected. We compare results on three well-known DCNs, Fattree [9], VL2 [22] and BCube [24].[2]

Running time. Table 2 shows the algorithm running time for constructing a probe matrix achieving 2-coverage and 1-identifiability. The strawman approach is our PMC algorithm without any optimizations. The last three columns contain results when the respective optimization is in place (in addition to the previous one(s)). The results show that PMC can efficiently select probe paths for very large DCNs. Specifically, without algorithm speedup, the computation time of PMC can be larger than 24 hours; after each optimization, the time decreases significantly and we can compute the probe matrix for Fattree(72), VL2(140,120,100) and BCube(8,4) within 18 seconds, 86 seconds and 70 seconds, respectively. We note that the running time in case of problem decomposition for VL2 and BCube is a bit longer than that of strawman. This is because decomposition does not apply to the two DCN topologies, but we need extra time to decide whether the matrix is decomposable.

Path number. Table 3 shows the number of selected paths with different α and β in different DCNs. Compared with the number of original paths in DCNs, PMC only selects a small percentage of paths. We can prove that the least number of paths for achieving 1-coverage and 1-identifiability is $k^3/5$ for any k-ary Fattree (Appendix B in [7]). Thus, a Fattree(64) DCN needs at least 52428 paths and our algorithm selects slightly more, *i.e.*,

[2] BCube is a server centric architecture and we treat servers as switches to run our algorithm.

Table 2: Algorithm running time (seconds) with $\alpha = 2, \beta = 1$ in different DCNs

DCNs	# of nodes	# of links	# of original paths	Strawman	Decomposition	Lazy update	Symmetry reduction
Fattree(12)	612	1296	184,032	231.458	5.216	0.506	0.126
Fattree(24)	4,176	10,368	11,902,464	> 24h	1381.226	23.254	0.280
Fattree(72)	99,792	279,936	8,703,770,112	> 24h	> 24h	> 24h	17.054
VL2(20, 12, 20)	1,282	1,440	70,800	22.030	23.126	0.77	0.253
VL2(40, 24, 40)	9,884	10,560	4,588,800	7387.412	7470.476	39.028	1.404
VL2(140,120,100)	424,390	436,800	4,938,024,000	> 24h	>24h	>24h	85.567
BCube(4, 2)	112	192	12,096	4.871	4.936	0.227	0.117
BCube(8, 2)	704	1,536	784,896	4050.776	4390.168	9.854	0.220
BCube(8, 4)	53,248	163,840	5,368,545,280	> 24h	> 24h	> 24h	69.778

Table 3: Number of selected paths with different (α,β)

DCNs	Original paths	Selected paths with (α, β)		
		(1, 0)	(1, 1)	(3, 2)
Fattree(32)	66,977,792	4,096	7,680	12,288
Fattree(64)	4,292,870,144	32,768	61,440	98,304
VL2(72,48,40)	107,371,008	864	1,440	2,640
VL2(128,96,80)	2,415,132,672	3,072	5,760	9,216
BCube (8,2)	784,896	1,712	2,016	2,832
BCube (8,4)	5,368,545,280	49,152	70,572	119,556

61440 paths. This implies that pingers under each selected ToR in the Fattree are only responsible for probing about 60 paths, much fewer than that of Pingmesh (about 2000-5000 paths). We also find that VL2 requires much fewer paths than Fattree and BCube. This is because VL2 has a much smaller number of links between switches (12288 links in VL2(128, 96,80)), as compared to Fattree (131072 links in Fattree(64)) and BCube (163840 links in BCube(8,4)).

Note that the number of selected path may change when the third optimization, based on topology *symmetry*, is in place. Our evaluation shows that the number of selected paths with symmetry reduction is very similar to that without symmetry reduction. This is consistent with the result in [30], and we hence omit the analysis.

Results for $\beta \geq 3$. The probe matrices we constructed above achieve at most 2-identifiability. For $\beta \geq 3$, the computation of PMC is not efficient in large DCNs. For the example of a 48-ary Fattree, computing a probe matrix achieving 3-identifiability requires at least 24 hours, even when we apply all speedup optimizations in §4.3. The fundamental reason is that the routing matrix **R** becomes much larger when the number of column increases from n to $\sum_{1 \leq i \leq \beta} C(n,i)$, by adding virtual links. However, surprisingly, we find that 2-identifiability is enough for loss localization in DCNs, as we will see in §6.4.

5 Loss Localization

5.1 Data Pre-processing

After collecting the probe data, the first step is to pre-process the data, removing outliers and normal cases. Severe packet losses could be caused by bad pingers and responders (*e.g.*, the server is down or was rebooting during probing, thus causing many false alarms [37]). Such outliers can be identified by keeping track of the status of servers using a watchdog service. In addition, a link normally has a regular low loss rate, *e.g.*, 10^{-4}–10^{-5}, due to transient congestion, bit errors, which should not be considered as failures [26]. To exclude such normal cases, we filter out paths with extremely low packet loss rates by setting a threshold on the number of packet losses in a period of time or on packet loss ratio (*e.g.*, 10^{-3} [26, 21]).[3] After pre-processing, the loss data that remain (in the form of $(path,\ number\ of\ losses)$) are likely manifest of network failures rather than noises.

5.2 Problem

Our fault localization problem is: given end-to-end packet loss observations, find the smallest set of faulty links that best explains the observations. This problem is NP-hard as it can be reduced to the NP-complete Minimum Hitting Set Problem [18]. Besides, we face two challenges not existed in previous work:

Much larger problem scale. Our study focuses on large-scale DCN networks, different from smaller networks investigated in the existing loss localization work [10, 18, 42]. At our problem scale, the existing algorithms are not fast enough (taking tens of seconds or even minutes) for real-time loss localization.

Different loss patterns. Network failures are mainly exhibited as two kinds of packet losses: full packet loss and partial packet loss, meaning that all or part of the packets traversing a link are dropped. Existing tomography techniques assume that if all links on a path are good, then the path is good [19]. This is not true in case of partial packet loss in data centers, *e.g.*, packet blackhole may lead to losses on a link only for a subset of paths using that link.

5.3 PLL Algorithm

Based on the *Tomo* algorithm in [18], we design an efficient Packet Loss Localization algorithm, *PLL*, to local-

[3] To avoid inaccuracy of the threshold approach, we can use statistical hypothesis testing to look at loss rates over time for noisy data filtering [27].

ize packet losses in DCNs (see [7] for more details). The basic idea of *PLL* is as follows.

Step 1: Divide the problem into a series of subproblems, by decomposing the probe matrix following the same steps discussed for decomposing the routing matrix in §4.3. For each subproblem, run the following steps.

Step 2: If all probe paths traversing a link experience no packet loss, we exclude the link. For the remaining links, we calculate a hit ratio for each link, *i.e.*, the ratio of the number of observed lossy paths through the link over the number of all probe paths using the link [34].

Step 3: We compute a score for each link as the number of lost packets that the link can explain, *i.e.*, if a link lies in the packet path, we say the link can explain the packet loss.

Step 4: Among those links whose hit ratios are larger than a preset threshold, we greedily select the link with the maximal score and remove those losses this link can explain.

Step 5: Repeat Step 3 and Step 4 until no loss remains unexplained.

PLL differs from *Tomo* mainly in handling partial packet losses, *i.e.*, we use a hit ratio threshold to filter suspected links. Setting the threshold requires network operator's experience and, if possible, by learning from real loss data. The analysis on setting this threshold is presented in [7] due to space constraint and we set it to 0.6 by default in our experiments.

We have compared performance of *PLL* and other existing loss localization methods (*e.g.*, *Tomo*, *SCORE* [34] and *OMP* [42]), and present the results in [7]. The results show that given the same probe matrix, *PLL* achieves 2% higher accuracy (defined as true positive ratio, *i.e.*, the percentage of bad links correctly identified as bad over all truly bad links), 2% lower false positive ratio (*i.e.*, the percentage of good links incorrectly identified as bad over all correctly and incorrectly identified links), and is an order of magnitude faster (*e.g.*, localizing failures within 1 second in a large DCN with 82944 links) than the other algorithms.

6 Implementation and Evaluation

6.1 Implementation

We run the controller on one Dell server (or it can run in a distributed fashion over multiple servers for large-scale networks). A watchdog service also runs on the server for monitoring the health of other servers and removing bad ones. The controller runs the PMC algorithm to recompute the probe matrix every 10 minutes, based on the current network topology from the watchdog service.[4] The computed probe matrix is divided into XML pinglist files for dispatching to pingers. A pinglist file contains file version, the pinger's IP address, IP addresses of responders, transport port numbers, the packet-sending interval and IP addresses of core switches. Our measurement shows that the controller can handle 4473 pinglist requests per second on average with maximal bandwidth consumption 688.56Mb/s using one core. Since pingers are deployed on a small number of servers (about 10% among all servers), the controller can support more than 100,000 pingers by slightly randomizing the time when pingers request for pinglists in each cycle.

Each pinger implements a communication module and a probing module. The communication module is responsible for connections with the controller and the diagnoser. It fetches the pinglist file from the controller by an HTTP GET request in every cycle (*i.e.*, 10 minutes). The probing module generates probe packets according to the pinglist and encapsulates them by IP-in-IP (§3.1). In our experiments, a pinger loops over a range of ports for each path, and emits several packets for every port. Each probe packet has an average size of 850 Bytes, carrying a specified DSCP value in the IP header to test different QoS classes [12]. If there is no response for a probe within 100ms, we mark it as a loss. A pinger repeatedly sends packets by looping through the paths in the pinglist for multiple times (for statistical accuracy), at the rate of 10 packets per second. Every 30 seconds, the pinger aggregates the probing results (*i.e.*, the number of packet losses and the number of packets sent on each probe path) into an XML file and sends it to the diagnoser by an HTTP POST request. The responder module runs in the userspace of all servers, which listens to a particular port, and upon packet arrival, it adds a timestamp and sends the packet back. The pinger and responder incur little overhead on servers, as we will see in §6.3.

The diagnoser is a Web server module running on the same server where the controller is in our experiments. It runs the PLL algorithm for fault localization once every half a minute, using collected probe results in the past 30 seconds. Given the limited number of servers in our testbed, we run a virtual machine to emulate a server.

6.2 Experiment Setup

We build a 4-ary Fattree testbed with 20 ONetSwitch [5, 29, 28], each equipped with FPGA-based hardware reconfigurable dataplane, four 1GbE ports and one dedicated management port. Though we do not require programmable switches in *deTector*, employing SDN switches facilitates our emulation of various failure cases that may happen in a real-world DCN. Specifically, we categorize all losses into three types:

[4] Once a link or a switch has failed, we remove related link(s) from the routing matrix to avoid selecting bad paths for probing. Note that it does not affect symmetry computation which only pre-runs once on the original DCN topology.

Full packet loss. We install OpenFlow rules with high priority to drop all packets coming from a particular port, to emulate a faulty link with full packet loss. To emulate a switch down case, we install rules to drop all packets at the switch.

Deterministic partial loss. Packets with certain features (*e.g.*, specific IPs, port numbers) may be dropped on a link deterministically, *e.g.*, in case of packet blackhole or misconfigured routing rules. To emulate such failures, we install rules on the switches to match and drop packets with certain headers.

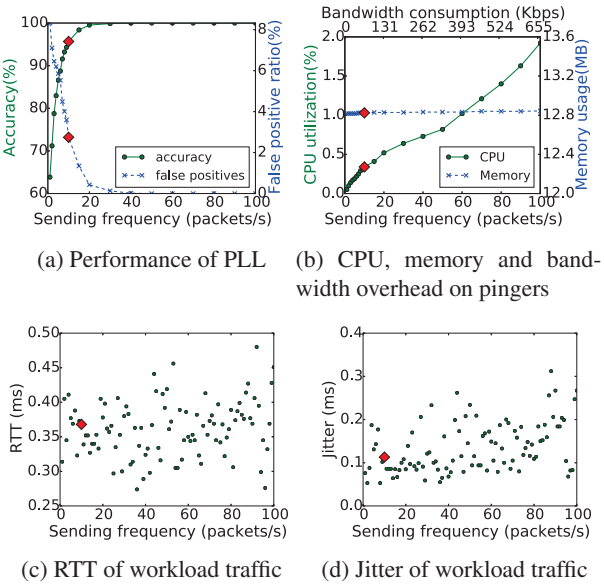
Random partial loss. Sometimes packets on a link are dropped randomly, as caused by bit flips, CRC errors, buffer overflow, etc. SDN switches do not support random packet dropping. To emulate such losses, we install rules on the switches to redirect all packets on an emulated bad link to the SDN controller, and the SDN controller drops the received packets with certain probability, following the pattern extracted from [12].

Due to no access to loss data in real-world data centers, we produce the above loss types according to the failure measurements in [20] and traffic measurements in [12]. Specifically, we set parameters such as link vs. switch failure percentage, link loss rates (ranging from 10^{-4} to 1), failure probabilities for switches in different tiers, all based on the above measurements. The loss distribution for links in different tiers is extracted from Fig. 3 in [12]. Aside from *deTector*, we also implement the probing modules of Pingmesh and NetNORAD on our testbed for performance comparison, as well as their failure localization tools, Netbouncer and fbtracert. Since we do not know some of their implementation details (*e.g.*, how data pre-processing is done), we implement those details in the same way across all three systems.

6.3 Performance

We first investigate how probing itself affects the whole DCN. We use realistic packet traces (including information such as packet header, timestamp) from a university data center [11] (mostly HTTP flows) to generate workload traffic in our testbed, where each server continuously replays flows based on the packet traces and sends them to a random receiver. We evaluate how our probing frequency (*i.e.*, the number of probes a pinger sends per second) affects the performance of the PLL algorithm, the overhead on the pinger, and RTT and jitter experienced by the workload traffic. In each minute of our experiment, we emulate a failure randomly picked among the three types of failures, with the failed switches or links randomly picked in the DCN. We run our experiment for 2000 minutes and obtain the average results.

Fig. 4 shows that a higher probe sending frequency leads to a higher accuracy and a lower false positive ratio (Fig. 4(a)), but causes higher CPU utilization and bandwidth consumption on pingers (Fig. 4(b)) as well as slightly larger fluctuation of the RTT (Fig. 4(c)) and jitter (Fig. 4(d)) experienced by the workload. We find that 10–15 probes per second is good enough since we can still achieve higher than 95% accuracy and a lower than 3% false positive ratio, while only consuming about 100Kbps bandwidth, 0.4% CPU and 13MB memory on each pinger. Besides, it does not introduce apparent delay and jitter variations for workload traffic. Note that the overhead of a responder is much smaller than a pinger because it resumes fewer tasks (*e.g.*, no communication with the controller and the diagnoser), and hence the results are omitted.[5] In all our experiments, the pinger sends 10 packets per second by default (*i.e.*, the red square in Fig. 4).



(a) Performance of PLL (b) CPU, memory and bandwidth overhead on pingers

(c) RTT of workload traffic (d) Jitter of workload traffic

Figure 4: Sensitivity test of sending frequency

We then compare the accuracy, false positive ratio and overhead among *deTector*, Pingmesh and NetNORAD. Since Pingmesh can not localize failures by itself, once it detects a suspected source-destination server pair, we use Netbouncer [4] to go through all possible paths between this server pair for loss localization. As for NetNORAD, similarly, we use fbtracert [3] to probe all possible paths between the suspected server pair. The interval of loss data collection is 30 seconds for three systems.

Fig. 5 shows the comparison when one failure is emulated in the testbed (the failure is randomly picked as in the previous experiment). The number of (ping and reply) probes in the figure includes probes sent for detection and probes for localization (if any) in each minute of the experiment. More probes indicate not only more

[5] Even when we place the pinger and responder on the same server, the overhead is negligible.

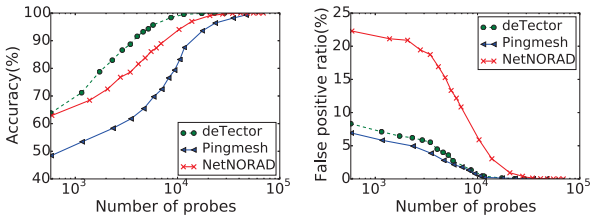


Figure 5: Accuracy and false positives of three monitoring systems with different number of probes per minute

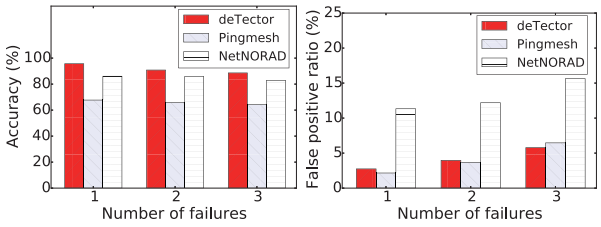


Figure 6: Results comparison with multiple failures

bandwidth consumption, but also higher CPU and memory usage. For *deTector*, we use a probe matrix with 1-identifiability and 3-coverage (since it is impossible to achieve 2-identifiability in a 4-ary Fattree). As we can see, *deTector* achieves high accuracy and a low false positive ratio with a much smaller number of probes, because *deTector* covers more types of losses (*e.g.*, low rate loss) and takes carefully planned paths. For instance, to achieve 98% accuracy and 1% false positives, *deTector*, NetNORAD and Pingmesh need to send 7200, 20700 and 35100 probes per minute, respectively. When the probe overhead is same (same number of probes sent per minute), the accuracy and false positive ratio achieved by *deTector* is better than those of NetNROAD; as compared to Pingmesh, the accuracy of *deTector* is much better, while the false positive ratio of Pingmesh is slightly smaller sometimes, since it possibly probes all paths.

Fig. 6 further shows the accuracy and false positive ratio with multiple failures, when the probe overhead is fixed to be the same, *i.e.*, 5850 probes per minute. *deTector* always achieves much better performance than Pingmesh and NetNORAD. Note that *deTector* also detects and localizes failures much faster than NetNORAD and Pingmesh (30 seconds in advance in our experiments), because *deTector* does not need any other diagnosis tools to send an additional round of probes for loss localization, while others do.

6.4 Simulation

We supplement our experimental evaluation with simulations, to investigate how identifiability of the probe matrix influences the accuracy of our failure localization, when running *deTector* in larger Fattree networks.

Table 4: Accuracy in a 18-radix Fattree, with probe matrices of different levels of coverage and identifiability

(α,β)	# of paths	Accuracy (%) with # of failed links				
		1	5	10	20	50
$(1,0)$	729	30.56	30.87	30.30	30.26	29.19
$(2,0)$	1485	58.43	57.43	57.08	56.81	57.11
$(3,0)$	2187	68.22	70.61	69.89	70.40	70.14
$(1,1)$	1269	94.74	93.37	94.21	93.43	90.29
$(1,2)$	1512	99.26	99.06	99.02	98.77	95.92
$(1,3)$	2349	99.63	99.63	99.67	99.62	98.07

We first vary α and β for probe matrix construction in an 18-radix Fattree. Table 4 shows that higher coverage and higher identifiability lead to higher accuracy, while the overhead (*i.e.*, the number of selected paths) does not increase much. Also, we find that identifiability is more effective and desirable than coverage for failure localization, since a 1-identifiability matrix increases the accuracy a lot (from one with 0-identifiability guarantee), with much less overhead than a 3-coverage probe matrix.

Note that further increasing the level of identifiability for $\beta > 1$ does not increase the accuracy much, and probe matrices achieving 1-identifiability can already lead to higher than 90% accuracy. According to the measurements in [12], less than 10% failure events (failures occurring concurrently) contain more than four failures and less than 1% failure events contain more than 20 failures. This implies that a probe matrix with 1-identifiability can guarantee higher than 93% accuracy for 90% failure events and 2-identifiability provides a 98% accuracy for 99% failure events.

The result is surprising but reasonable: Since we use a number of optimizations (§4.3) to reduce the size of the routing matrix, the PMC algorithm in fact achieves β'-identifiability (where β' is larger than β used in the algorithm) for the whole probe matrix, rather than β-identifiability computed for each small probe matrix (corresponding to a small network topology). Therefore, *deTector* may fail to localize all failures only if more than β failures appear in a small topology, which occurs with relatively low probability. This shows that using a probe matrix with a low level of identifiability guarantee is good enough to identify a much larger number of concurrent failures.

In addition, by examining the failure events that *deTector* fails to localize with a low identifiability probe matrix but can identify using a high identifiability matrix, we find that higher identifiability achieves better results only when the number of simultaneously failed links is very large. Such a failure event with many concurrent link failures is usually triggered by a common bug in practice (*e.g.*, 180 links fail simultaneously due to scheduled maintenance to multiple aggregation switches [20]),

Table 5: Fault localization performance with probe matrix of 2-identifiability in a 48-ary Fattree

# of failed links	1	5	10	20	50
Accuracy (%)	98.95	98.99	98.98	98.93	98.87
False positive (%)	0.01	0.02	0.02	0.02	0.02
False negative (%)	1.05	1.01	1.02	1.07	1.13

and thus those faulty links are spatially clustered. In such cases, operators can locate the failure spot effectively according to the positions of most failed links.

We further examine the fault localization accuracy, false positive and false negative (bad links incorrectly identified as good) ratios achieved using a probe matrix of 2-identifiability in a 48-ary Fattree. Table 5 shows that the false positive and false negative ratios remain in a very low level. In particular, the false positive rate is extremely low ($< 1\%$), which is desirable in practice [18].

The false negatives are mainly caused by losses of extremely low loss rate and intermittent losses which may happen at longer intervals (than 1 minute) [23]. Since it takes longer time to expose these losses, we can further reduce false negatives by examining loss measurements in larger time windows, *e.g.*, 10 minutes.

7 Discussions

Packet entropy. *deTector* tries to increase packet entropy (*i.e.*, different packet patterns) by varying IP addresses, port numbers and DSCP values, to cover as many failures as possible. However, our implementation uses IP-in-IP encapsulation for source routing, and hence the range of destination IP addresses is somewhat limited. In addition, since we use UDP for network probing, *deTector* may not be able to detect failures related to other protocols, *e.g.*, misconfigured TCP parameters [26]. Adopting other source routing solutions and adding more protocols to increase packet entropy are part of our future work.

Loss diagnosis. While *deTector* can localize where packet drops occur, it does not know what causes the drops, *e.g.*, software bugs, misconfigured rules or bursty traffic. This is a common deficiency of existing monitoring systems, since network diagnosis is rather complex. However, it is possible to distinguish full losses, deterministic partial losses, random partial losses and losses due to congestion, to narrow down the diagnosis scope (*e.g.*, using machine learning approaches), since they exhibit different loss characteristics. We consider this as a promising future direction to explore.

Beyond *deTector*. As opposed to probe-based solutions like *deTector*, there are some recent efforts on embedding metadata in the packet header to trace packet path for network debugging (*e.g.*, CherryPick [46], PathDump [47]). Our technique can be applied to reduce the overhead involved in these approaches, *i.e.*, only packets traversing those paths computed by the PMC algorithm need to carry routing information in the packet headers.

8 Related Work

Probe design. Many existing work (*e.g.*, [14, 18, 43, 33, 27]) exploit logs on switches, or utilize multicast or network coding for network probing. Instead, we treat each switch as a blackbox, and adopt a topology-aware end-to-end probing approach. Some studies [16, 40, 23] estimate loss rates of all links, while we aim at identifying bad links (*i.e.*, failure spots). Zeng *et al.* [48] and Nicolas *et al.* [23] propose monitoring solutions for backbone networks that do not apply in DCNs due to scalability, and the main difference lies in probe matrix design.

Fault localization. Our goal of accurately identifying faulty links falls squarely in the area of binary network tomography. Tomography algorithms such as Sherlock [10], Tomo [18], GREEDY [35], SCORE [34] and OMP [42] do not work well for DCNs due to their problem scales and loss characteristics. Our *PLL* algorithm is built on these work and conquers their limitations.

DCN monitoring. Our work mainly differs from existing monitoring systems such as Pingmesh [26] and NetNORAD [37] in the design of probe matrix. We argue that loss detection and localization must be coupled together to localize more failures (*e.g.*, transient failures) in real time with low overhead. Carefully designed probe matrix is the key to achieve them. LossRadar [39] is a switch-based solution but it requires programmable switches. Dapper [44] and Zipkin [8] are distributed tracing systems to gather timing data for root-cause analysis.

9 Conclusion

deTector is a real-time, low-overhead and high-accuracy monitoring system for large-scale data center networks. At its core is a carefully designed probe matrix, constructed by a scalable greedy path selection algorithm with minimized probe overhead. We also design an efficient failure localization algorithm according to different patterns of packet losses. Our analysis, testbed experiments and large-scale simulations show that *deTector* is highly scalable, practically deployable with low overhead, and can localize failures with high accuracy in near real time.

Acknowledgments We thank Xiaowei Wu for his help with algorithm design. This work was supported by National Key Research and Development Program of China 2016YFB0800101, NSFC 61672425, NSFC 61628209, and Hong Kong RGC grants HKU 718513, 17204715, 17225516, C7036-15G (CRF).

References

[1] VXLAN. https://tools.ietf.org/html/rfc7348, 2014.

[2] NVGRE. https://tools.ietf.org/html/rfc7637, 2015.

[3] Fbtracert. https://github.com/facebook/fbtracert, 2016.

[4] Microsoft Netbouncer. https://www.youtube.com/watch?v=nfEOEKlInK8, 2016.

[5] ONetSwitch. http://www.onetswitch.org/index, 2016.

[6] deTector project. https://github.com/yhpeng-git/deTector, 2017.

[7] deTector technical report. https://github.com/yhpeng-git/deTector/blob/master/documentation/technical_report.pdf, 2017.

[8] Zipkin. http://zipkin.io, 2017.

[9] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A scalable, commodity data center network architecture. In *Proc. of ACM SIGCOMM* (2008).

[10] BAHL, P., CHANDRA, R., GREENBERG, A., KANDULA, S., MALTZ, D. A., AND ZHANG, M. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proc. of ACM SIGCOMM* (2007).

[11] BENSON, T. Data set for IMC 2010 data center measurement. http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html, 2010.

[12] BENSON, T., ANAND, A., AKELLA, A., AND ZHANG, M. Understanding data center traffic characteristics. In *Proc. of ACM SIGCOMM* (2010).

[13] BRODIE, M., RISH, I., AND MA, S. Optimizing probe selection for fault localization. In *Proc. of the 12th International Workshop on Distributed Systems: Operations and Management (DSOM)* (2001).

[14] CASTRO, R., COATES, M., LIANG, G., NOWAK, R., AND YU, B. Network tomography: recent developments. *Statistical Science 19*, 3 (2004), 499–517.

[15] CHEN, K., GUO, C., WU, H., YUAN, J., FENG, Z., CHEN, Y., LU, S., AND WU, W. Generic and automatic address configuration for data center networks. In *Proc. of ACM SIGCOMM* (2010).

[16] CHEN, Y., BINDEL, D., SONG, H., AND KATZ, R. H. An algebraic approach to practical and scalable overlay network monitoring. In *Proc. of ACM SIGCOMM* (2004).

[17] DARGA, P. T., SAKALLAH, K. A., AND MARKOV, I. L. Faster symmetry discovery using sparsity of symmetries. In *Proc. of the 45th Annual Design Automation Conference (DAC)* (2008).

[18] DHAMDHERE, A., TEIXEIRA, R., DOVROLIS, C., AND DIOT, C. Netdiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data. In *Proc. of the 3rd ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2007).

[19] DUFFIELD, N. Network tomography of binary network performance characteristics. *IEEE Transactions on Information Theory 52*, 12 (November 2006), 5373–5388.

[20] GILL, P., JAIN, N., AND NAGAPPAN, N. Understanding network failures in data centers: measurement, analysis, and implications. In *Proc. of ACM SIGCOMM* (2011).

[21] GOVINDAN, R., MINEI, I., KALLAHALLA, M., KOLEY, B., AND VAHDAT, A. Evolve or die: High-availability design principles drawn from Google's network infrastructure. In *Proc. of ACM SIGCOMM* (2016).

[22] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. VL2: a scalable and flexible data center network. In *Proc. of ACM SIGCOMM* (2009).

[23] GUILBAUD, N., AND CARTLIDGE, R. Localizing packet loss in a large complex network. https://www.nanog.org/meetings/nanog57/presentations/Tuesday/tues.general.GuilbaudCartlidge.Topology.7.pdf, 2013.

[24] GUO, C., LU, G., LI, D., WU, H., ZHANG, X., SHI, Y., TIAN, C., ZHANG, Y., AND LU, S. BCube: a high performance, server-centric network architecture for modular data centers. In *Proc. of ACM SIGCOMM* (2009).

[25] GUO, C., WU, H., TAN, K., SHI, L., ZHANG, Y., AND LU, S. DCell: a scalable and fault-tolerant network structure for data centers. In *Proc. of ACM SIGCOMM* (2008).

[26] GUO, C., YUAN, L., XIANG, D., DANG, Y., HUANG, R., MALTZ, D., LIU, Z., WANG, V., PANG, B., CHEN, H., ET AL. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proc. of ACM SIGCOMM*, 2015.

[27] HERODOTOU, H., DING, B., BALAKRISHNAN, S., OUTHRED, G., AND FITTER, P. Scalable near real-time failure localization of data center networks. In *Proc. of the 20th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)* (2014).

[28] HU, C., YANG, J., GONG, Z., DENG, S., AND ZHAO, H. DesktopDC: setting all programmable data center networking testbed on desk. *ACM SIGCOMM Computer Communication Review 44*, 4 (2015), 593–594.

[29] HU, C., YANG, J., ZHAO, H., AND LU, J. Design of all programmable innovation platform for software defined networking. In *In Proc. of the 4th Open Networking Summit (ONS)* (2014).

[30] HU, S., CHEN, K., WU, H., BAI, W., LAN, C., WANG, H., ZHAO, H., AND GUO, C. Explicit path control in commodity data centers: Design and applications. In *Proc. of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2015).

[31] ISARD, M. Autopilot: automatic data center management. *ACM SIGOPS Operating Systems Review 41*, 2 (April 2007), 60–67.

[32] JYOTHI, S. A., DONG, M., AND GODFREY, P. Towards a flexible data center fabric with source routing. In *Proc. of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)* (2015).

[33] KANDULA, S., MAHAJAN, R., VERKAIK, P., AGARWAL, S., PADHYE, J., AND BAHL, P. Detailed diagnosis in enterprise networks. In *Proc. of ACM SIGCOMM* (2009).

[34] KOMPELLA, R. R., YATES, J., GREENBERG, A., AND SNOEREN, A. C. IP fault localization via risk modeling. In *Proc. of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2005).

[35] KOMPELLA, R. R., YATES, J., GREENBERG, A., AND SNOEREN, A. C. Detection and localization of network black holes. In *Proc. of IEEE INFOCOM* (2007).

[36] LAPUKHOV, P. Configuring IPSLA. `http://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst4500/12-2/44sg/configuration/guide/Wrapper-44SG/swipsla.html`.

[37] LAPUKHOV, P. Network debugging at scale. `https://www.nanog.org/sites/default/files/Lapukhov_Move_Fast_Unbreak.pdf`, 2016.

[38] LESKOVEC, J., KRAUSE, A., GUESTRIN, C., FALOUTSOS, C., VANBRIESEN, J., AND GLANCE, N. Cost-effective outbreak detection in networks. In *Proc. of the 13th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)* (2007).

[39] LIÚ, Y., MIAO, R., KIM, C., AND YUÚ, M. LossRadar: Fast detection of lost packets in data center networks. In *Proc. of the 12th ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2016).

[40] MA, L., HE, T., LEUNG, K. K., TOWSLEY, D., AND SWAMI, A. Efficient identification of additive link metrics via network tomography. In *Proc. of the 33rd IEEE International Conference on Distributed Computing Systems (ICDCS)* (2013).

[41] PATEL, P., BANSAL, D., YUAN, L., MURTHY, A., GREENBERG, A., MALTZ, D. A., KERN, R., KUMAR, H., ZIKOS, M., WU, H., ET AL. Ananta: cloud scale load balancing. In *Proc. of ACM SIGCOMM* (2013).

[42] PATI, Y. C., REZAIIFAR, R., AND KRISHNAPRASAD, P. Orthogonal matching pursuit: Recursive function approximation with applications to wavelet decomposition. In *Proc. of the 27th Asilomar Conference on Signals, Systems and Computers (ACSSC)* (1993).

[43] SHARMA, G., JAGGI, S., AND DEY, B. Network tomography via network coding. In *Proc. of the 3rd Information Theory and Applications Workshop (ITA)* (2008).

[44] 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., Google, Inc., 2010.

[45] SINGH, A., ONG, J., AGARWAL, A., ANDERSON, G., ARMISTEAD, A., BANNON, R., BOVING, S., DESAI, G., FELDERMAN, B., GERMANO, P.,

ET AL. Jupiter rising: A decade of Clos topologies and centralized control in Google's datacenter network. In *Proc. of ACM SIGCOMM* (2015).

[46] TAMMANA, P., AGARWAL, R., AND LEE, M. Cherrypick: Tracing packet trajectory in software-defined datacenter networks. In *Proc. of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)* (2015).

[47] TAMMANA, P., AGARWAL, R., AND LEE, M. Simplifying datacenter network debugging with pathdump. In *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016).

[48] ZENG, H., MAHAJAN, R., MCKEOWN, N., VARGHESE, G., YUAN, L., AND ZHANG, M. Measuring and troubleshooting large operational multipath networks with gray box testing, msr-tr-2015-55. Tech. rep., 2015.

Pricing Intra-Datacenter Networks with Over-Committed Bandwidth Guarantee

Jian Guo[1], Fangming Liu[1]*, Tao Wang[1], John C.S. Lui[2]
[1]*Key Laboratory of Services Computing Technology and System, Ministry of Education, School of Computer Science and Technology, Huazhong University of Science and Technology*
[2]*The Chinese University of Hong Kong*

Abstract

Current IaaS clouds provide performance guarantee on CPU and memory but no quantitative network performance for VM instances. Our measurements from three production IaaS clouds show that for the VMs with same CPU and memory, or similar pricing, the difference in bandwidth performance can be as much as 16×, which reveals a severe price-performance anomaly due to a lack of pricing for bandwidth guarantee. Considering the low network utilization in cloud-scale datacenters, we address this by presenting SoftBW, a system that enables pricing bandwidth with over commitment on bandwidth guarantee. SoftBW leverages usage-based charging to guarantee price-performance consistency among tenants, and implements a fulfillment based scheduling to provide bandwidth/fairness guarantee under bandwidth over commitment. Both testbed experiments and large-scale simulation results validate SoftBW's ability of providing efficient bandwidth guarantee, and show that by using bandwidth over commitment, SoftBW increases 3.9× network utilization while incurring less than 5% guarantee failure.

1 Introduction

Cloud computing enables enterprises and individuals to access computing resources based on their demands with a simple pay-as-you-go model. Large numbers of cloud tenants are multiplexed in the same datacenter (DC) infrastructure, where they can also get isolated computing resources via virtual machines (VMs). In current IaaS clouds, CPU performance (represented by vCPUs) and memory performance (represented by GB) are both quantifiable metrics. However, datacenter network bandwidth, which can severely impact the job completion time of network-intensive applications, has not been standardized as one of the VM performance metrics.

To reveal the network performance in IaaS clouds, we measured the intra-datacenter bandwidth of VMs from three popular cloud platforms, i.e., Google Compute Engine (GCE), Amazon EC2 and Aliyun ECS. The measurement indicates a severe *price-performance anomaly* in current clouds: 1) for VMs from different clouds, whose CPU/memory and prices are the same, the difference in network performance can be as much as 16 times; 2) for VMs in the same cloud, the average bandwidth of a cheaper VM can surpass the bandwidth of an expensive one; 3) for a single VM, the network performance at different time is varying and highly unpredictable. Hence, due to a lack of bandwidth performance guarantee and a corresponding pricing strategy, tenants deploying network intensive applications can hardly obtain the performance in agreement with which they pay for.

To price network bandwidth with a performance guarantee, a practical solution should not only satisfy the bandwidth requirements of tenants, but should also achieve efficient resource utilization to benefit providers' profit. Given that current cloud-scale datacenters have a low network utilization (99% links are less than $<$ 10% loaded [1]), we propose to allow over commitment for bandwidth guarantee as well as provide usage-based pricing for tenants. For example, a provider can co-locate VM_1 with 8 Gbps bandwidth and VM_2 with 4 Gbps bandwidth on a server with 10 Gbps bandwidth. When both VMs are transmitting traffic continuously, VM_1 and VM_2 get 6.7 Gbps and 3.3 Gbps bandwidth, respectively. For a given billing cycle, they both get a discount in proportional to the fulfillment ratio, i.e., $5/6$. To validate the feasibility of bandwidth over commitment, we model the failure rate of bandwidth guarantee based on the datacenter traffic traces in [2, 3], and show that we can control

This work was supported in part by the National 973 Basic Research Program under Grant 2014CB347800, and in part by NSFC under Grant 61520106005. The work of John C.S. Lui is supported in part by the GRF 14205114. (*Corresponding author: Fangming Liu)

the expected failure rate within an acceptable level by using proper over commitment ratio (§2).

However, pricing bandwidth guarantee under over commitment condition is a challenging task. Existing work on bandwidth guarantee, which focuses on guaranteeing tenants' bandwidth requirements under sufficient bandwidth condition, can hardly achieve the target:

- They do not provide a usage-based pricing for the corresponding bandwidth allocation techniques to achieve price-performance consistency.
- The state-of-the-art bandwidth allocation solution using either static or dynamic rate limit cannot provide performance guarantee under bandwidth over commitment.
- Existing solution verifies bandwidth guarantee by using long-lived flows, but ignores the performance degradation of short flows when using periodically rate limit.

To address the challenges, we propose SoftBW, a solution that enables pricing datacenter networks via software-defined bandwidth allocation, aiming to achieve: price-performance consistency, over commitment tolerance and short flow friendly (§4). SoftBW realizes a usage-based charging by monitoring a guarantee fulfillment, which is a ratio of the achieved bandwidth to the committed bandwidth guarantee, on each billing cycle. The pricing strategy, when combined with our bandwidth allocation, ensures that tenants paying higher unit price can achieve higher network performance.

SoftBW implements a fulfillment-based scheduling (§5) to simultaneously provide minimum bandwidth guarantee under sufficient physical bandwidth condition, and guarantee VM-level fairness when the physical bandwidth is constrained. By applying a dynamic guarantee for long-lived traffic, SoftBW can utilize the idle bandwidth to reduce the total bandwidth guarantee, which further reduces the guarantee failures under bandwidth over commitment. SoftBW's implementation uses a software virtual switch at each server, and introduces only 5.1% CPU overhead and less than 1.9 μs latency for 10 Gbps data transmission. Testbed experiments validate SoftBW's ability to efficiently provide bandwidth guarantee, as well as having 2.8$\times$ to 4.5$\times$ improvement on the completion time of short flows, as compared with existing rate-limit based approaches. In large scale simulation, we find that using bandwidth over commitment can increase $\sim$ 3.9$\times$ network utilization, while maintain the average failure of bandwidth guarantee under 5%. In summary, the contributions of this paper consist of:

- By measuring the intra-DC bandwidth of VMs from three cloud platforms, we reveal the severe price-performance anomaly among different clouds, due to a lack of pricing for quantitative bandwidth performance.
- We validate the feasibility of pricing bandwidth guarantees with over commitment in current multi-tenant datacenters by proposing a usage-based charging model and a fulfillment-based scheduling algorithm.
- We develop SoftBW, a system that implements the proposed pricing and scheduling, and show that SoftBW can provide efficient bandwidth/fairness guarantee for both long-lived traffic and short flows under bandwidth over commitment in testbed experiments.

2 Background and Motivation

2.1 VM Bandwidth in Public Cloud

We measure the intra-DC bandwidth among different instance types[1] from four selected datacenters: N. Virginia in US East (Amazon), N. California in US West (Amazon), Asia East (Google), Beijing in China (Alibaba). Each throughput is collected 12 times a day, lasting 5 minutes on every 2 hours. The maximum and average throughput of different instances is shown in Table 1. During the measurement, the CPU utilizations of all VMs are less than 100% of a single core, which indicates that the bottleneck is on the network bandwidth. The instance types with the same network performance are merged into one group. We find that while VM rate-limit is commonly used, the *limited bandwidth* and *direction of rate-limit* for VMs are quite different among different providers.

Rate-limit upperbound. Alibaba ECS simply provides the same rate-limit for all VMs at 520 Mbps. For Amazon EC2, the sharing strategy of the two datacenters sees no obvious distinctions. As expected, the performance corresponds to the description of VM instances (excluding the ones with 10 Gbps dedicated bandwidth) in EC2, which falls into three levels, i.e., low to moderate, moderate (300 Mbps) and high (1 Gbps). The throughput of "low to moderate" is fairly unstable and can be as large as about 3 Gbps. Although Google does not claim the network performance of VMs, most of them outperform the VMs in EC2 in both average and maximum throughput, which has three levels, i.e., 1 Gbps, 2 Gbps and 4 Gbps. For VMs with 4 or more vCPUs, we do not find any obvious rate-limit. The stable throughput varies from 1 Gbps up to 5 Gbps.

Rate-limit direction. There are two rate-limit strategies in these clouds: source based rate-limit for egress traffic (Google and Alibaba), and rate-limit for both

[1]The measurement covers all shared-core and standard instances in GCE (cloud.google.com), all general purpose instances in EC2 (aws.amazon.com), and all #vCPUs in Alibaba ECS.

EC2	Low to Moderate	Moderate	High
Low to Moderate	0.84/3.00	0.29/0.30	0.79/1.01
Moderate	0.29/0.30	0.29/0.29	0.29/0.29
High	0.55/1.01	0.29/0.30	0.97/1.16

GCE	Low	Moderate	High	Highest
Low	0.59/1.01	0.66/1.01	0.64/1.01	0.64/1.00
Moderate	0.76/2.00	1.97/2.00	1.99/2.00	1.98/2.00
High	0.65/3.36	2.78/3.24	2.73/3.20	3.03/4.00
Highest	0.86/4.93	3.32/4.05	3.56/3.95	4.36/5.09

Table 1: The average/max throughput of VM-to-VM traffic (Gbps) in Amazon EC2 and Google Compute Engine (GCE) datacenters. The left column and head row are source and destination VM, respectively. [1]

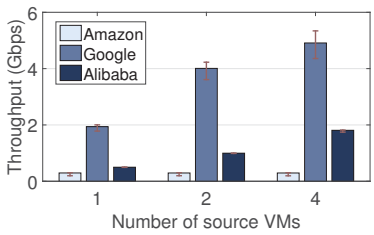


Figure 1: Comparing the average, max, min receiving rate of VM with different numbers of sending VMs.

ingress and egress traffic (Amazon). In Figure 1, we validate this by showing the receiving rates of VMs with different number of sending VMs. The instance types we studied in three clouds are guaranteed to have the nearest performance in CPU and memory: 1 vCPU, 3.75 GB memory for GCE, 1 vCPU, 3.75 GB memory for EC2, and 1 vCPU, 4 GB memory for ECS. The receiving rates of VMs in EC2 do not increase with more sending VMs, which indicates that the ingress bandwidth of VM is limited. GCE and ECS have no rate-limit for ingress traffic of VMs unless they are congested by physical bandwidth. Hence, the maximal rates can reach at about 5 Gbps and 2 Gbps with 4 sending VMs, respectively.

Price-performance anomaly. For VMs in different clouds, which have the same allocated resource (i.e., vCPUs and memory) and pricing, the bandwidth performance is significantly different. For example, the VMs in Figure 1 have 1 vCPU and 3.75 GB memory, but the gap in network performance is as much as 6 to 16 times. As a result, some cloud providers may miss a golden opportunity to achieve higher competitiveness in the market due to a lack of quantitative bandwidth performance. In fact, as indicated by the missing of bandwidth performance for VMs, currently the maximal throughput is not guaranteed by the provider since we observe significant variation in throughput during one day.

2.2 Why Over Commitment is Rational?

To validate the feasibility of over selling network bandwidth in clouds, we start with modeling the datacenter traffic based on existing measurement work [2, 3]. The ratio of over commitment of a server is defined as the ratio of the sum of guaranteed bandwidth C_B to the physical bandwidth C at this server, denoted by $\delta = C_B/C$. When over commitment is involved, the risk of guarantee failure caused by resource over commitment should be considered.

Suppose the server hosts n homogeneous VMs with the same bandwidth guarantee, whose traffic is independent identically distributed. As indicated by [2], the traffic demand on an edge port approximates an exponential distribution with the probability density satisfying $f(x) = \alpha e^{-\alpha x}$, where x is the traffic demand and $1/\alpha$ is the average traffic demand. Note that $1/\alpha$ can be obtained by tracking the average network utilization of a VM. In a δ over committed server, where the bandwidth guarantees of VMs exceed the physical bandwidth, the guarantee fails when the total traffic demands exceed the physical bandwidth C, namely, $\sum x_i > C$, where x_i represents the demand of VM i, $i \in [1,n]$.

Let Θ denote the domain that subjects to $\sum x_i > C (x_i > 0)$ for those n VMs. Then the probability of failure P_n follows the joint probability distribution that every x_i locates in Θ, which is an n-dimensional integral

$$P_n = \int \dots \int_{\Theta} \prod \alpha e^{-\alpha x_i} \mathrm{d}x_1 \dots \mathrm{d}x_n. \quad (1)$$

Solving above equation, P_n can be expressed as

$$P_n = e^{-\alpha C} \sum_{i=1}^{n} \frac{(\alpha C)^{i-1}}{(i-1)!}. \quad (2)$$

Let ρ be the average network utilization of a host server without over commitment, then $\alpha = n/\rho C_B$. Figure 2 shows the impact of over commitment on the failure rate with 16 VMs. We maintain the server network utilization as 10% and 15% (the value can be adjusted according to the network utilization by providers). As we can see, the failure rate is less than 5% if using 6.9× OC for 10% average network utilization, or using 4.6× OC for 15% average network utilization.

The simple analysis shows that when the access bandwidth is over committed, the expected failure rate can be controlled within an acceptable level by using proper ratio of over commitment according to the average utilization. Although in practical situation, a VM's traffic may not follow an ideal exponential distribution, the over commitment is still worth deployment in a large scale,

[1] There is no description for VM network performance in GCE. The measured performance can be divided into 4 groups: low (f1-micro, g1-small), moderate (n1-standard-1), high (n1-standard-2), and highest (n1-standard-4/8/16)). The inter-VM throughput of Alibaba is omitted as the strategy is similar to GCE.

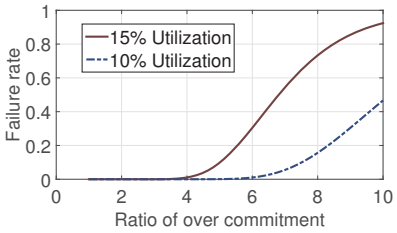


Figure 2: The impact of bandwidth over commitment on the failure rate of bandwidth guarantee.

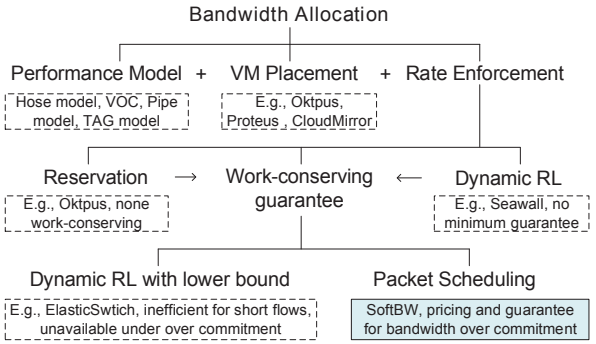


Figure 3: Technical position of SoftBW in bandwidth allocation: rate enforcement with packet scheduling.

and the VMs still get a minimum guarantee of C_B/δ in the worst case.

2.3 Why not Existing Solutions?

Achieving bandwidth guarantee for VMs in datacenters needs to address three key tasks [4] as shown in Figure 3: a performance model that specifies the tenants' bandwidth requirements, a VM placement mechanism that allocates VMs to the servers with sufficient physical bandwidth, and a rate control algorithm that dynamically controls the rates of VMs to improve bandwidth utilization. Current rate-limit (RL) based solution can meet the basic requirements of bandwidth guarantee [5, 6], i.e., minimum guarantee, proportional sharing, and work-conserving. However, they do not provide a pricing strategy and can hardly address the challenges on bandwidth over commitment in datacenters.

First, the rate-limit based solution does not provide guaranteed performance metrics for bandwidth over commitment. They work like TCP for aggregated VM-to-VM traffic, i.e., keep increasing and multiplicatively decrease when congested, so as to provide VM-level fairness. To achieve bandwidth guarantee for a VM, the rate limitation needs to stay above the minimum guarantee, thus the limitation of traffic from other VMs will reduce, and their traffic that exceeds the guaranteed bandwidth can be restricted [5]. This policy assumes that the access links at end-hosts are not over-subscribed. When the total bandwidth guarantee exceeds the physical bandwidth on a server, for example, three VMs each with 500 Mbps minimum bandwidth guarantee are co-located on a server with 1 Gbps bandwidth, the minimum rate-limit of each VM (500 Mbps) is held upon a fair share (333 Mbps), thus becoming unavailable. To avoid this, one should tell whether the total traffic demand will exceed the physical bandwidth in the next update of rate limitations, and decide whether the limitations need to be lower-bounded. However, predicting traffic demands at $\sim$ 50 ms granularity is extremely hard for datacenter traffic. An efficient rate enforcement mechanism is needed to guarantee fairness when the physical bandwidth is over committed while providing minimum bandwidth under sufficient bandwidth condition.

Second, periodically rate limit degrades the performance of VM-to-VM traffic when the traffic is an aggregation of massive short flows. To achieve work-conserving [7], the unused bandwidth of idle VMs is allocated to other VMs rather than be statically reserved. Hence, the rate limitations of VMs need to be updated at an interval of tens of milliseconds [5], which is longer than the completion times of most of short flows. During this period, if the traffic of a VM has fully utilized the allocated bandwidth, the newly arrived short flows will compete with existing traffic and cause a short-term congestion. It not only delays the transmission of short flows, but also degrades the performance of existing flows. One may address this by using more fine-grained rate control. However, frequent changes of rate limitation, especially sudden decrease in rate limitation, can cause significant fluctuations to the underlying TCP flows. For a tenant, it is unacceptable that the use of idle bandwidth is at the cost of degrading the performance of short flows. Hence, a packet-level solution that can "take back" the paid bandwidth quickly is more suitable for bandwidth pricing.

3 Fulfillment Abstraction

3.1 Access Bandwidth vs. Congested Links

Our bandwidth allocation focuses on end-based rate enforcement, as shown in Figure 3. The choice comes from the fact that today's production datacenters see rapid advances in achieving full bisection bandwidth [8, 9], and the providers have a growing concern about the over committed access bandwidth on each server rather than the aggregation and core level. By leveraging the software virtual switch at each server, the cost of implementation can be reduced and the scale of rate control is limited to the number of VMs on each server. Our design as-

sumes the datacenter fabric to be a non-blocking switch [10, 11, 7], and our main focus is to schedule the traffic at the virtual ports connected to VMs.

3.2 Guarantee Fulfillment

Our work aims at enforcing the bandwidth at the VM-level, and providing pricing schemes for bandwidth guarantee. This requires an abstraction that not only provides a performance metric for bandwidth sharing under bandwidth over commitment, but also serves as a quota for charging. To this end, we propose the concept of *guarantee fulfillment* to express tenants' bandwidth performance. The fulfillment is defined as the ratio of VM x's rate r_x to its promised bandwidth guarantee B_x:

$$F_x = \frac{r_x}{B_x}. \quad (3)$$

As the fulfillment takes a bandwidth guarantee as a baseline, it is complementary to existing network model for expressing tenants' bandwidth requirements. For example, B_x can be the VM bandwidth in a Virtual Cluster [10] model. We define the bandwidth guarantee for each VM since it can be better adapted to current per-VM based charging in cloud. Note that the abstraction can also be extended to the VM-to-VM bandwidth in a Tenant Application Graph model, if we setup a virtual queue for each VM-to-VM pair at both source and destination servers.

Fulfillment for scheduling. The performance guarantee for the tenants relies on maintaining fairness among VMs' fulfillments. When bandwidth is sufficient, the fairness of fulfillments means that for any VM x, y, $F_x = F_y > 1$, and the VMs have minimum bandwidth guarantee since $r_x > B_x$. If the bandwidth is over committed, the VMs may have $F_x < 1$ when the total traffic demand exceeds the physical bandwidth. By maintaining the same fulfillment, network proportionality can be achieved, i.e., $r_x : r_y = B_x : B_y$ for any VM x, y. Thus the worst case performance under δ over commitment will be no less than B_x/δ and B_y/δ.

Fulfillment for pricing. As a charging quota for providers, the fulfillment indicates how much of the paid bandwidth is actually obtained by a tenant. The bandwidth is charged according to the fulfillment of VMs measured on a billing cycle (e.g., per second), where a discount is applied based on the actual usage. The billing cycle is similar to the minimum period for charging in current clouds, e.g., GCE use per-minute billing for VM instances. To price the bandwidth under over commitment, two tasks should be done: First, a model to estimate the failure of guarantee as a service commitment for the failure rate in the SLA (similar to the monthly uptime percentage in EC2 SLA [12]) (§2.2). Second, a *fulfillment-based pricing function* that guarantees tenants paying higher prices can achieve higher performance.

4 SoftBW Design

SoftBW enables pricing intra-DC bandwidth under over commitment by scheduling packets to satisfy VMs' bandwidth requirements and charging based on the actual bandwidth used by VMs. Specifically, our design targets at the following goals:

- **Price-performance consistency.** Tenants paying higher price should achieve proportionally higher bandwidth performance. Tenants can not achieve higher performance by lying about their bandwidth requirements.
- **Over commitment tolerance.** The system should simultaneously provide minimum bandwidth guarantee when physical bandwidth is sufficient, and guarantee fairness when their minimum guarantees exceed the physical bandwidth.
- **Short flow friendly.** The performance of short flows with bandwidth guarantee should not be degraded when the physical bandwidth is occupied by other traffic.

4.1 Architectural Overview

SoftBW uses a Software-Defined Networking architecture where each host server deploys a virtual switch that can be controlled by centralized controllers. As Figure 4 shows, SoftBW leverages a centralized master to manage the business in the control plane, and enforces bandwidth allocation using distributed agents in the data plane. The system consists of two functions: (i) pricing the bandwidth based on the measured fulfillment from the traffic monitor (§4.3), and (ii) enforcing the requirements of bandwidth guarantee by using packet scheduling in the virtual switch (§5). The two functions both have decoupled modules in SoftBW master and SoftBW agent nodes.

SoftBW master maintains the requirements (bandwidth paid by tenants) and fulfillments of VMs at a logically centralized server. The information is used by cloud providers to define their charging models. SoftBW agent leverages the virtual SDN switch at each server to schedule packets from per-VM queues by obtaining the requirements from the controller. The scheduling algorithm works in a round robin manner where the VMs with less fulfillment can be preferentially scheduled. The overhead of virtual SDN switch, which is determined by the number of VMs on each server, does not increase as we scale up the number of servers in datacenters.

SoftBW works as follows. When a VM is launched and connects to a port of the virtual switch, the data plane generates an asynchronous message to the control plane, which notifies the connection of a node. SoftBW master can capture the port connected to the VM, and sends

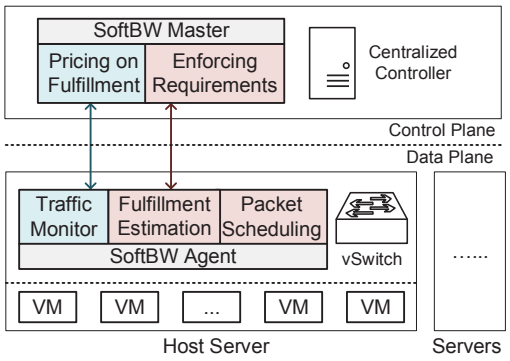


Figure 4: System overview: the data plane schedules packets to satisfy the VMs' bandwidth requirements, and the control plane manages pricing using measured fulfillments.

out the corresponding bandwidth guarantee to the agent. Afterwards, the agent creates a queue for this VM and adds a flow table entry to match and enqueue the packets of the VM. In the lifetime of the VM, the traffic monitor in agent monitors the rates of VMs, and feeds the rates back periodically to compute the price and update the guarantee values of VMs which require the dynamic bandwidth guarantee. Note that the fulfillment for pricing, which is measured at a billing cycle, is isolated from the fulfillment for scheduling at the packet level.

4.2 Performance Metrics

SoftBW allows the provider to take advantage of low network utilization in datacenters to oversell the physical network bandwidth. One important note is that different applications need different kinds of guarantees. For example, delay-tolerant applications like background backup, whose completion times are only related to the total throughput during the period of backup. Therefore, a cloud provider does not need to provide a strict rate guarantee for the entire duration of the backup job. Instead, the bandwidth can be allocated to other VMs which are running real-time jobs, and then compensate the backup jobs when more bandwidth is available. As a result, applications which require strict rate guarantee and applications which require deadline guarantee can both be satisfied.

Differentiated performance metrics. We now propose three different network performances, which allow the bandwidth guarantee can be dynamically allocated, thus to reduce the risk of guarantee failure as well as increase the network utilization under over commitment situation.

- *Strict guarantee* provides the real-time minimum bandwidth guarantee for a VM, which is denoted by a dedicated rate B.
- *Dynamic guarantee* ensures the total deliverable traffic of a VM during a specific time period (e.g., time to deadline). The dynamic guarantee is denoted by a tuple (S,T), where S is the total traffic size and T is the desired transmission time.
- *Fairness guarantee* offers fair VM-level fairness for sharing the residual bandwidth, which is left by VMs with strict or dynamic guarantees, among all VMs.

Note that dynamic guarantee can be satisfied by an average rate of $b' = S/T$ Mbps. However, instead of guaranteeing this average rate for the whole duration T, we vary the bandwidth guarantee according to the traffic loads in datacenters. We first assign an initial minimum guarantee S/T for the VM. After a period of t_0 seconds, there might be s' Mb remaining data on that VM, which should be transmitted in $t' = T - t_0$ seconds. Then we update the guarantee b' as s'/t' Mbps (which is called the expected guarantee), and periodically repeat this update. As a result, the guarantee is dynamically adjusted according to the available bandwidth. If there is residual bandwidth on the server, the VM can utilize it and reduce the guarantee in the next update. As a result, the total bandwidth guarantee on a server is reduced, and the probability of guarantee failure also decreases. If the bandwidth is not guaranteed for some periods, the VM can increase the guarantee and still finish the transmission within the expected time.

However, if a VM with dynamic guarantee does not send traffic at the beginning of transmission, the time to finish transmission t' decreases and the traffic size s' remains the same. For this case, the expected guarantee $b' = s'/t'$ will increase and even exceed the initial guarantee. Hence, we need to maintain the dynamic guarantee under the initial guarantee S/T, and only provide fairness guarantee after T.

4.3 Pricing Model

Usage-based charging. With bandwidth over commitment, the throughput of VMs may not achieve the guaranteed bandwidth (i.e., guarantee failure). To address this, SoftBW charges bandwidth according to the actual bandwidth usage. Suppose the unit price of strict bandwidth guarantee B_i is P_s. The VM with B_i bandwidth guarantee is charged $P_s \cdot B_i \cdot F_t$ at billing period t, where F_t is the fulfillment measured at period t. Traffic that exceeds B_i will be charged the same as the pricing of fairness guarantee (P_f), since it only gets a fair sharing. For example, in a billing cycle, if the throughput of a VM with 100 Mbps strict guarantee is 150 Mbps, the price will be $100P_s + 50P_f$. For dynamic guarantee, the unit price of bandwidth guarantee P_d relies on the average bandwidth guarantee $B_j = S/T$. The cost at period t is $P_d \cdot B_j \cdot F_t$. As dynamic guarantee may reduce the

Guarantee	Performance	Price
Strict	Bandwidth B	$r_t \cdot (1+B/C)P_0$
Dynamic	Data size S, time T	$r_t \cdot (1+S/TC)\beta P_0$
Fairness	VM-level fairness	$r_t \cdot r_t/C \cdot \beta P_0$
Best effort	No bandwidth guarantee	Free

Table 2: The price for bandwidth guarantee at a billing cycle: r_t is the measured rate of a VM.

failure rate in bandwidth guarantee under low network utilization situation, we set $P_d = \beta P_s$ ($\beta < 1$) to encourage tenants to use dynamic guarantee for massive delay-tolerant data transmission.

Performance-price consistency. However, for usage-based pricing, the tenants can declare higher bandwidth than their requirements to achieve higher performance under the same price, since the transmission time of the same size of data is proportionally reduced. For example, when transmitting 1 Gb data, using 100 Mbps bandwidth will cost 10 seconds, while using 200 Mbps bandwidth only costs 5 seconds. Both situations cost $1000P$, where P denotes the price of using 1 Mbps for 1 seconds. Hence, to keep performance-price consistency, the unit price of higher bandwidth guarantee should also be higher.

We use a non-decreasing pricing function for bandwidth guarantee, where $P_s = P_0(1+B/C)$, $P_d = (1+S/TC)\beta P_0$ (C represents the physical bandwidth, and P_0 is a constant price). Fairness guarantee has the lowest unit price $P_f = r_t/C \cdot \beta P_0$, which is always less than P_d. This way, tenants using strict guarantee will buy the lowest possible bandwidth according their requirements, and avoid unnecessary data transmission. For tenants using dynamic guarantee, the bandwidth guarantee becomes zero when finishing transmission of the declared data S. Thus, under-declaring the data size will not benefit the performance. In fact, they will transmit data as fast as possible, because their transmission costs will be cheaper if exceeding the expected bandwidth B_j. This also benefits the provider: the idle bandwidth is utilized and the dynamic guarantee decreases, thus more bandwidth can be used for other guaranteed traffic.

5 Fulfillment-based Scheduling

SoftBW applies a packet level scheduling in the agent to share bandwidth under over commitment based on the fulfillment abstraction in §3. SoftBW agent leverages the generalized processor sharing model [13] to serve the queue in a weighted round robin manner [14]. Thus, each queue gets a share of the bandwidth, and the fairness among VMs can be obtained even when bandwidth is constrained. For bandwidth guarantee, instead of using periodical rate measurement, the agent measures the transmission time of each packet, and only estimates whether the bandwidth guarantee of the VM is satisfied.

Our scheduling, the *estimation of fulfillment* and the *scheduling of packets*. In a round robin scheduling, the key task is to decide whether the packets at the head of queues should be transmitted at each round. We set up a timer to record the time-to-send (tts) for the packet at the head of the queue, which indicates the expected time point to transmit this packet if we want to meet the bandwidth guarantee. Then the scheduler decides whether the packet should be transmitted by comparing tts against the current time. Since the fulfillment of a queue decreases when it is waiting to be scheduled, the goal of fulfillment estimation is to update the time-to-send for the queues after each transmission.

5.1 Estimation of Fulfillment

For a queue, each time a packet (p_n) is transmitted, we calculate the inter-departure time between this packet and the last transmitted packet (p_{n-1}), denoted as τ. Let p_{size} denote the size of the packet (p_n). If the bandwidth guarantee for this queue is B, then the fulfillment can be expressed as $F = \frac{p_{size}/\tau}{B}$. Thus, for a VM whose bandwidth guarantee is not satisfied ($F < 1$), we can derive the following equation

$$\Delta\tau = \tau - \frac{p_{size}}{B} > 0, \tag{4}$$

where $\Delta\tau$ is the difference between the inter-departure time and the expected time of transmitting a packet with B. Since this difference in transmitting time means that the VM's rate is either larger ($\Delta\tau < 0$) or less ($\Delta\tau > 0$) than the guaranteed bandwidth, the inter-departure time should be accumulated in every update, so as to reduce the rate that is above the guaranteed bandwidth, as well as increasing the rate that is under the guaranteed bandwidth. Thus, $\overline{\Delta\tau} \leftarrow \overline{\Delta\tau} + \Delta\tau$. We maintain $\overline{\Delta\tau}$ in the interval $[-\tau_{max}, \tau_{max}]$ so that $\overline{\Delta\tau}$ will not infinitely decrease when bandwidth exceeds the guarantee, nor increase when bandwidth can not satisfy the guarantee.

Each time when $\overline{\Delta\tau}$ of a queue is re-calculated, we update the tts for this queue:

- If $\overline{\Delta\tau} \geq 0$, the bandwidth guarantee of the VM is not satisfied. Thus, we set tts to 0, to allow the scheduler to dequeue a packet from the queue.
- If $\overline{\Delta\tau} < 0$, the rate of the VM exceeds B. Then, the variable tts of the VMs is set to p_{size}/B ahead of current time:

  $$tts = t_{current} + \frac{p_{size}}{B}, \tag{5}$$

  which notifies the scheduler if a packet is transmitted before this time tts, the VM will exceeds the bandwidth guarantee (B).

- If $\overline{\Delta\tau}$ of a queue changes from positive to negative, it implies that the sending rate of the corresponding VM just exceeds its bandwidth guarantee, then we set

$$tts = t_{current} + \frac{p_{size}}{B} - \overline{\Delta\tau}. \quad (6)$$

  $\overline{\Delta\tau}$ is a compensation for the rate, since the VM's rate in previous round is less than the bandwidth guarantee.

5.2 Scheduling of Packets

Before scheduling a packet, the scheduler first compares the current time $t_{current}$ against the tts of a queue. There are three conditions to consider:

- If $tts = 0$, then the rate of the VM is below the bandwidth guarantee and the scheduler dequeues the packet at the head of the queue.
- If $0 < tts \leq t_{current}$, the scheduler has just missed the expected transmission time. If the packet is transmitted at the current time, the VM's rate will not exceed the guaranteed bandwidth. Hence, the scheduler can transmit a packet from this queue.
- If $tts > t_{current}$, then the VM will exceed the bandwidth guarantee after we send a packet. For this case, the scheduler will check the status of the physical bandwidth and only sends a packet if there is any residual bandwidth on this server.

Work-conserving. Similar to queues of VMs, the status of the physical bandwidth is maintained by calculating the difference ($\Delta\tau_c$) between the inter-departure time and the expected time of transmitting a packet with the maximal physical rate, after transmitting a packet from any queue. When $\Delta\tau_c > 0$, which indicates the physical bandwidth is not fully utilized, scheduler can still transmit packets from queues that have exceeded the bandwidth guarantee. This way, the residual bandwidth of the host server can be allocated if there is unsatisfied traffic demand, thus the bandwidth sharing is work-conserving.

Performance guarantee. The round robin scheduler can preferentially transmit the packets from VMs whose bandwidth guarantee is not satisfied. As a result, the rates of these VMs can quickly increase even when the physical bandwidth are taken by other traffic. For example, when a VM with a bandwidth guarantee starts to send traffic on a fully utilized access link, the newly arrived packets can be transmitted at each round as the VM's fulfillment is less than 1 and tts is 0. Note that this also benefits the performance of the short flows, since they can transmit packets without waiting for other VMs to decrease their rates. For fairness guarantee, we need to set a small bandwidth guarantee for the queues, so that their traffic will not be blocked when the bandwidth is fully utilized by guaranteed traffic.

6 Evaluation

In this section, we evaluate the performance of SoftBW from the following aspects:

- *Performance guarantee*: We validate that SoftBW can achieve stable bandwidth guarantee and at the same time, maintain fairness among VMs even when bandwidth is over committed.
- *Fast allocation*: We validate that SoftBW has the property of small convergence time (~ 10 ms) in the presence of highly bursty traffic, and $2.8\times \sim 4.5\times$ improvement on the completion time for short flows as compared with the rate-limit approach.
- *Overhead*: We analyze the overhead of SoftBW and find that SoftBW adds less than 1.9 μs transmission delay to the TCP's RTT, and maintains less than 5.1% CPU overhead under 10 Gbps transmission.
- *Over commitment*: We examine the impact of bandwidth over commitment on resources sharing and show that the provider can possibly increase average $3.9\times$ network utilization while maintaining the average failure rate within 5% in our simulation.

6.1 Evaluation Setup

We first perform testbed experiments to evaluate SoftBW's performance on bandwidth guarantee (§6.2) and the overhead of scheduling (§6.3). We then use simulation to study the impact of over commitment on large scale (§6.4).

Testbed. The testbed consists of 14 servers. Each server has an Intel Xeon E5-2670 2.6 Ghz CPU (8 physical cores with hyper-threading) and an Intel 82580 Gigabit NIC connected to a 1 GbE switch port. The servers run Linux 2.6.32 kernel, among which one acts as the controller with OpenDaylight and the others host servers with KVM and Open vSwitch (OVS). Each VM has a `virtio` NIC with `vhost-net` enabled, connecting to a `tap` device attached to an OVS bridge. We compare SoftBW with an existing rate-limit based bandwidth allocation [5], represented as ES (ElasticSwitch).

Simulator. We simulate a 2,000-server datacenter with full bisection bandwidth. Each server connects to the switch with a 1 Gbps link. The strict guarantee and the initial expected guarantee for dynamic guarantee are both 250 Mbps. Thus without over commitment, each server can deploy 4 VMs. As we focus on network-intensive applications, the simulator only considers network bandwidth, and allocates bandwidth at 1s interval.

Workload. In the simulation, we use two different traffic loads: 1) For strict guarantee, the traffic demand of a VM follows a exponential distribution around a mean of 250ρ Mbps on each time slot. 2) For dynamic guarantee, the data size in a VM follows a exponential dis-

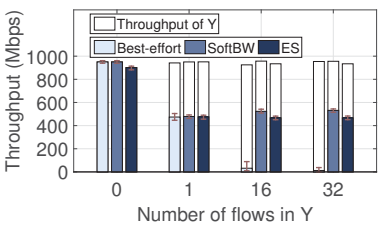


Figure 5: Throughput of X with increasing number of flows in Y. Both VMs have 450 Mbps guarantees.

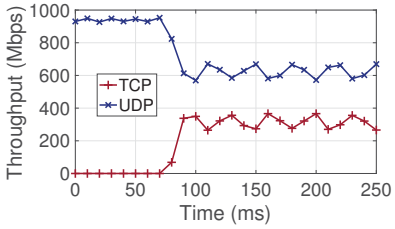


Figure 6: The convergence process of X when Y sends a bursty UDP flow on the same congested link.

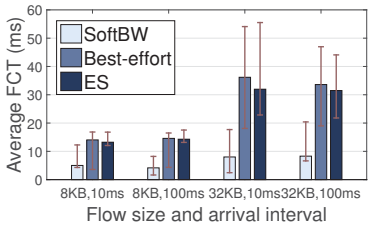


Figure 7: Guarantee for short flows: SoftBW improves the average flow completion time (FCT).

tribution with a mean of $250\rho \cdot T_{max}$ Mbits, and the start time follows a Poisson process with N_d/T_{max} arrival rate. (T_{max}: simulation time, N_d: #VMs with dynamic guarantee.) The source and destination of traffic on each VM are chosen uniformly at random. Thus the expected network utilization without over commitment will be ρ.

Parameters. The evaluation chooses 1 second as the time interval of updating the dynamic bandwidth guarantee. Note that it is a suitable setting when compared with VM lifetime. As TCP flows can achieve at most 940 Mbps in our testbed, the maximum guarantee without over commitment is set as 90% of 1 Gbps physical bandwidth on each server.

6.2 Bandwidth Guarantee and Efficiency

Bandwidth guarantee. We co-locate two sender VMs on one server: VM X connects to a receiver with one TCP flow, and VM Y connects to N receivers, each with one flow. Both VM X and Y have 450 Mbps strict bandwidth guarantee. We vary N, the number of receivers of Y, and show the throughput of VM X in Figure 5. The left bar in each group represents the throughput without guarantee, and X suffers unfairness when Y has more flows. With SoftBW's scheduling, X is guaranteed with a rate of $\sim$450 Mbps when Y has multiple flows, and utilizes the entire link when Y has no traffic. This verifies SoftBW's work-conserving property and the ability of enforcing bandwidth guarantee under aggressive bandwidth competitions.

Convergence process. We show SoftBW's adaption to sudden traffic changes by quantifying the convergence process of long flows. When VM X (with 600 Mbps guarantee) sends traffic with a long-lived flow to a remote receiver and becomes stable, VM Y (with 300 Mbps guarantee) starts to generate bursty UDP traffic to another receiver with 800 Mbps sending rate. Figure 6 shows the throughput of X and Y measured at the receiving end. When the traffic in Y arrives, it consumes the bandwidth in around 10 ms, which demonstrates SoftBW's fast convergence on re-allocating the utilized bandwidth. Due to TCP's rate control, we observe fluctuations at around 10 ms timescale, however, the average throughput measured by every 100 ms is stable , which is sufficient for usage-based charging.

Short flows. Since short flows' durations are too short to fully utilize the guaranteed bandwidth, we quantify SoftBW's guarantee for short flows by examining the completion time of these flows when competing with existing long flows. Figure 7 illustrates the scenario where VM Y (450 Mbps bandwidth guarantee) is continuously sending traffic, and VM X (450 Mbps bandwidth guarantee) generates short flows on the same congested link. The short flows are of 8 KB/32 KB in size, and the flow inter-arrival times are 10 ms/100 ms. Without scheduling, the increase of flow rate relies on creating a congestion on the link which notifies the existing flows to adjust their rates (best-effort), or rate-limits those flows to decrease their sending rate (ES). These adjustment may take a long time for a short flow to acquire the necessary bandwidth resource. When enabling SoftBW, the packets from short flows can quickly obtain the bandwidth, since X's fulfillment is less than Y and so X's packets will be scheduled without delay at each round. As a result, the flows in X are guaranteed to have a higher average rate, thus the completion time is $2.8\times\sim4.5\times$ shorter than that of rate-limiting or best effort packet scheduling.

Over commitment. We set up an over committed scenario where three VMs each with 450 Mbps strict guarantee are sharing a 1 Gbps link. In the worst case, when all VMs are send traffic continuously, the total traffic demand exceeds 1 Gbps and their bandwidth can not be guaranteed. Figure 8 shows the rates of VMs when the ratio of the number of flows in each VM is 1 : 1 : 2. Since SoftBW uses per-VM queue, each VM obtains about 300 Mbps bandwidth, and the fairness among VMs is guaranteed irrespective of the flows in VMs. However, since the rate-limit based guarantee policy relies on limiting the rate beyond the minimum guarantee, it can not maintain fairness under this condition where the rate of each VM is less than the bandwidth guarantee, thus VM with more flows receives more bandwidth.

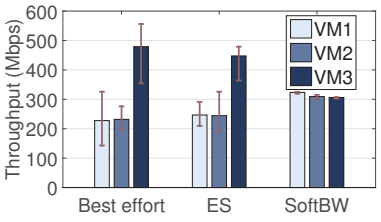


Figure 8: SoftBW provides fairness among VMs' throughput under bandwidth over commitment, where each VM has 450 Mbps guarantee.

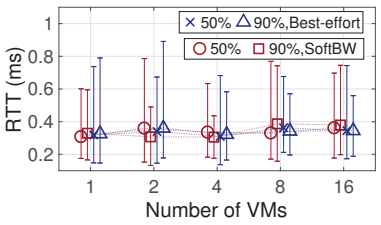


Figure 9: SoftBW does not add latency to RTT under 50% and 90% UDP traffic loads as compared to best-effort manner.

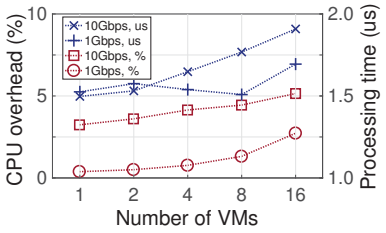


Figure 10: SoftBW's CPU overhead and processing time of each packet with different numbers of VMs on each server.

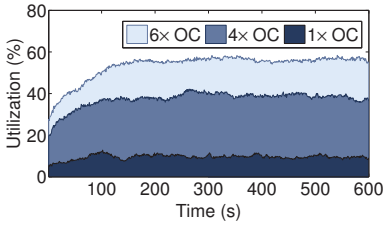


Figure 11: The network wide utilization in 600 s simulation under 1×, 4× and 6× OC.

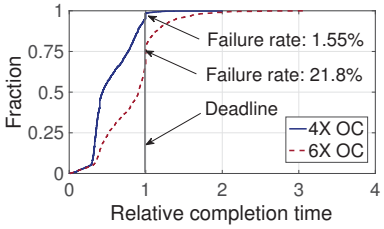


Figure 12: The relative completion time of data transmission using dynamic guarantee under 4× and 6× OC.

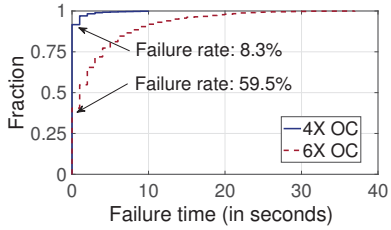


Figure 13: The failure time (in seconds) of VMs using strict guarantee under 4× and 6× OC.

6.3 Overhead Analysis

We evaluate SoftBW's latency overhead in comparison to the best effort manner without any scheduling. As we focus on network intensive applications, the packets from traffic generator have a size of MTU (1500 Bytes), which can achieve 10 Gbps throughput with only one CPU core. Hence, network is the only bottleneck in the experiments. For TCP delay, we leave a 10% gap between the maximum workloads and the physical bandwidth to reduce the impact of link congestion on RTT. The number of VMs is capped by the vCPUs on each server, i.e., 16.

Latency. Figure 9 shows the RTTs between two VMs on different servers, with 50% and 90% UDP traffic loads. As the traffic load and the number of VMs increase, we see no obvious increase of latency in RTT (with traffic less than 1 Gbps). The fluctuation of RTT is also small enough to maintain the stability of TCP flows. We attribute this to the high efficiency of OVS's software tunneling supported by today's powerful processors. At the same time, SoftBW's scheduling, which only adds at most five steps of floating point computation for each packet, will not be a bottleneck for packet transmission as compared to the VM-to-VM latency ($\sim 350\,\mu$s in the same availability zone of EC2).

CPU overhead. Figure 10 shows the CPU overhead and processing time of each packet with SoftBW. We conduct traffic by sending data from VMs to their host server, where the total throughput is rate limited under 10 Gbps by TC. As expected, the CPU overhead increases as we increase the total throughput and the number of VMs on each server. The maximum throughput reaches ~ 9 Gbps and the overhead with 16 VMs is only 5.1%. Considering that the access bandwidth in current data center are often 10 Gbps, this overhead is acceptable for providers. The processing time of each packet in the scheduling is about 1.5 μs. Due to the contention of CPU, the processing time increases as the number of VMs increases. This latency in scheduling is much less than TCP RTT, hence will not degrade the performance of underlying TCP flows.

6.4 Bandwidth Over Commitment

When bandwidth is over committed, the guarantee failure happens under two conditions: First, for a VM with strict guarantee, traffic demand is not satisfied when it is less than the bandwidth guarantee. Second, for dynamic guarantee, the traffic is not finished before the deadline. The simulator runs 600 s with $\rho = 15\%$, where we record the rate and transmission time of each VM.

Network utilization. Figure 11 shows the network wide utilization under different over commitment. The average utilization without over commitment is about 9.5%. When bandwidth is 4× OC, there is a remarkable improvement on the average utilization by 3.9×, from 9.5% to 37.4%. Using more aggressive OC (6×) can further increase the network utilization, however, as we will

show, also brings significant failures to the bandwidth or deadline guarantee.

Guarantee failures. Figure 12 shows the relative completion times (in percentile of guaranteed completion time) of data transmission with dynamic guarantee. Figure 13 shows the failure time of VMs with strict guarantee. With 4× OC, 98.4% VMs finish transmission before their deadlines under dynamic guarantee. Only 1.6% VMs fails, and the worst performance is 2.0× longer than the deadline. For VMs with strict guarantee, only 8.4% of them experience guarantee failure. The total failure duration of each VM is also very low, among which the longest one is 10 s during 600 s simulation. As a result, when the average network utilization is around 10%, it is feasible for providers to use 4× OC for bandwidth guarantee at a large scale, because there is little chance to encounter insufficient bandwidth, and the worst performance with guarantee failure is also acceptable. Thus the provider can consider to compensate the tenants for the guarantee failures at a low cost. When the over commitment increases to 6×, we can observe obvious guarantee failure for both strict guarantee and dynamic guarantee, whose failure rate are 59.5% and 21.8%, respectively. Hence, it is not suitable for this traffic load. The simulation validates the feasibility of bandwidth over commitment in multi-tenant clouds, and provides a solution to estimate the proper over commitment ratio by tracking the network utilization.

7 Related Work

Bandwidth allocation. The first piece of work focuses on bandwidth reservation in datacenters [10, 15, 16, 4]. By proposing performance models and VM allocation mechanisms, they allocate the VMs to servers which can meet the performance requirements. Another policy is to slice the physical bandwidth according to the traffic demand of VMs using rate-control [7, 17, 18]. The two solutions are complementary to each other, since the bandwidth of VMs can be re-shaped after allocation.

Faircloud [7] presents the bandwidth requirements of bandwidth allocation problem and develops traffic slicing strategies for proportional sharing. NetShare [19] achieves proportional bandwidth sharing among different tenants by using weighted fair queues in switches. SoftBW uses virtual switches in the hypervisor, hence, the traffic with bandwidth guarantee will not be limited by the number of hardware queues in the switches.

Seawall [20] leverages end-based rate limit to achieve VM-level weighted max-min fairness. This policy can be extended for bandwidth guarantee with a lower-bounded rate-limit for VMs [5, 21]. ElasticSwitch [5] partitions the bandwidth guarantees of VMs to VM-pairs, and achieves minimum guarantee using a TCP-Cubic based rate control for VM-to-VM traffic. EyeQ [11] measures rate every 200 μs at the receivers, and uses this information to enforces the rate of senders to achieve minimum guarantee. eBA [22, 23] uses the feedback of link utilization from switches to control the rate of senders. The rate-limiting based solution assumes that the total bandwidth guarantee is less than the physical bandwidth, and is not suitable for the cloud providers to over commit their bandwidth.

Bandwidth pricing. Usage-based pricing model is widely used in current IaaS clouds [24]. Recent proposals have studied the economical impact of cloud resource pricing on system design and providers' revenue [25, 26, 27]. We target at providing price-performance consistency, and improving providers' revenue is included in our future work. [28] answers the question of how users should bid for cloud spot instances, which aims at saving costs for users. [29] discusses dynamic pricing for inter-datacenter traffic, rather than intra-datacenter network bandwidth. The solutions are not suitable for our situation, since their pricing for bandwidth does not involve over commitment on bandwidth guarantee.

Scheduling/congestion control. SoftBW's scheduling framework is based on previous round robin scheduling such as, CBQ [30], Fair queueing [13], and adds mechanisms to enforce fairness of fulfillments. There are also a number of works , which use scheduling (e.g., [31, 32, 33]) or new congestion control (e.g., [34, 35, 36])) to improve the performance of flows and reduce the latency in datacenters. Virtual congestion control [37, 38] can help to deploy new congestion control in the hypervisors without changing VM network stack. Our work is complementary to these works, which focus on the performance at flow level, while we provide bandwidth guarantee and pricing for VMs.

8 Conclusion

In this paper, we presented SoftBW, a solution that enables pricing bandwidth for VMs in IaaS datacenters, by providing efficient bandwidth/fairness guarantee with bandwidth over commitment. SoftBW's over commitment on bandwidth is rational, since the failure rate of bandwidth guarantee can be controlled to a low level by conservatively choosing the over commitment ratio based on the network utilization. SoftBW's design is easy to be implemented, since it uses software virtual switches at each server to schedule VMs' traffic and can be centrally controlled. SoftBW applies a usage-based charging, which is deployable for current charging model in multi-tenant clouds, thus giving a feasible solution for providers to realize quantified bandwidth performance and pricing for cloud VM instances.

References

[1] 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.

[2] BENSON, T., AKELLA, A., AND MALTZ, D. A. Network traffic characteristics of data centers in the wild. In *ACM IMC* (2010).

[3] KANDULA, S., SENGUPTA, S., GREENBERG, A., PATEL, P., AND CHAIKEN, R. The nature of data center traffic: measurements & analysis. In *ACM IMC* (2009).

[4] LEE, J., TURNER, Y., LEE, M., POPA, L., BANERJEE, S., KANG, J.-M., AND SHARMA, P. Application-driven bandwidth guarantees in datacenters. In *ACM SIGCOMM* (2014).

[5] POPA, L., YALAGANDULA, P., BANERJEE, S., MOGUL, J. C., TURNER, Y., AND SANTOS, J. R. Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing. In *ACM SIGCOMM* (2013).

[6] GUO, J., LIU, F., ZENG, D., LUI, J. C., AND JIN, H. A cooperative game based allocation for sharing data center networks. In *IEEE INFOCOM* (2013).

[7] POPA, L., KUMAR, G., CHOWDHURY, M., KRISHNAMURTHY, A., RATNASAMY, S., AND STOICA, I. Faircloud: Sharing the network in cloud computing. In *ACM SIGCOMM* (2012).

[8] GREENBERG, A., HAMILTON, J., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D., PATEL, P., AND SENGUPTA, S. Vl2: a scalable and flexible data center network. In *ACM SIGCOMM* (2009).

[9] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication* (New York, NY, USA, 2008), SIGCOMM '08, ACM, pp. 63–74.

[10] BALLANI, H., COSTA, P., KARAGIANNIS, T., AND ROWSTRON, A. Towards predictable datacenter networks. In *ACM SIGCOMM* (2011).

[11] JEYAKUMAR, V., ALIZADEH, M., MAZIÈRES, D., PRABHAKAR, B., KIM, C., AND GREENBERG, A. Eyeq: practical network performance isolation at the edge. In *USENIX NSDI* (2013).

[12] Amazon ec2 service level agreement. http://aws.amazon.com/ec2/sla/.

[13] SHREEDHAR, M., AND VARGHESE, G. Efficient fair queueing using deficit round robin. In *ACM SIGCOMM Computer Communication Review* (1995).

[14] SHREEDHAR, M., AND VARGHESE, G. Efficient fair queueing using deficit round robin. *SIGCOMM Comput. Commun. Rev. 25*, 4 (Oct. 1995), 231–242.

[15] GUO, C., LU, G., WANG, H., YANG, S., KONG, C., SUN, P., WU, W., AND ZHANG, Y. Secondnet: A data center network virtualization architecture with bandwidth guarantees. In *ACM CoNEXT* (2010).

[16] XIE, D., DING, N., HU, Y., AND KOMPELLA, R. The only constant is change: incorporating time-varying network reservations in data centers. In *ACM SIGCOMM* (2012).

[17] GUO, J., LIU, F., LUI, J. C. S., AND JIN, H. Fair network bandwidth allocation in iaas datacenters via a cooperative game approach. *IEEE/ACM Trans. Networking 24*, 2 (Apr. 2016), 873–886.

[18] GUO, J., LIU, F., TANG, H., LIAN, Y., JIN, H., AND LUI, J. C. Falloc: Fair network bandwidth allocation in iaas datacenters via a bargaining game approach. In *IEEE ICNP* (2013).

[19] LAM, T., AND VARGHESE, G. Netshare: Virtualizing bandwidth within the cloud. Tech. rep., UCSD, 2009.

[20] SHIEH, A., KANDULA, S., GREENBERG, A., KIM, C., AND SAHA, B. Sharing the data center network. In *USENIX NSDI* (2011).

[21] RODRIGUES, H., SANTOS, J., TURNER, Y., SOARES, P., AND GUEDES, D. Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks. In *USENIX Workshop on I/O Virtualization* (2011).

[22] GUO, J., LIU, F., HUANG, X., LUI, J. C., HU, M., GAO, Q., AND JIN, H. On efficient bandwidth allocation for traffic variability in datacenters. In *IEEE INFOCOM* (2014).

[23] LIU, F., GUO, J., HUANG, X., AND LUI, J. C. S. eba: Efficient bandwidth guarantee under traffic variability in datacenters. *IEEE/ACM Trans. Netw. 25*, 1 (Feb. 2017), 506–519.

[24] ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A. D., KATZ, R., KONWINSKI, A., LEE, G., PATTERSON, D., RABKIN, A., STOICA, I., AND ZAHARIA, M. A view of cloud computing. *Commun. ACM 53*, 4 (Apr. 2010), 50–58.

[25] NIU, D., FENG, C., AND LI, B. Pricing cloud bandwidth reservations under demand uncertainty. In *ACM SIGMETRICS Performance Evaluation Review* (2012).

[26] WANG, H., JING, Q., CHEN, R., HE, B., QIAN, Z., AND ZHOU, L. Distributed systems meet economics: pricing in the cloud. In *USENIX HotCloud* (2010).

[27] XU, H., AND LI, B. A study of pricing for cloud resources. *ACM SIGMETRICS Performance Evaluation Review* (2013).

[28] 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* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 71–84.

[29] JALAPARTI, V., BLIZNETS, I., KANDULA, S., LUCIER, B., AND MENACHE, I. Dynamic pricing and traffic engineering for timely inter-datacenter transfers. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference* (New York, NY, USA, 2016), SIGCOMM '16, ACM, pp. 73–86.

[30] FLOYD, S., AND JACOBSON, V. Link-sharing and resource management models for packet networks. *IEEE/ACM TON* (1995).

[31] AL-FARES, M., RADHAKRISHNAN, S., RAGHAVAN, B., HUANG, N., AND VAHDAT, A. Hedera: Dynamic flow scheduling for data center networks. In *NSDI* (2010).

[32] ALIZADEH, M., YANG, S., SHARIF, M., KATTI, S., MCKEOWN, N., PRABHAKAR, B., AND SHENKER, S. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM* (2013).

[33] CHOWDHURY, M., ZHONG, Y., AND STOICA, I. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 443–454.

[34] 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* (New York, NY, USA, 2010), SIGCOMM '10, ACM, pp. 63–74.

[35] JANG, K., SHERRY, J., BALLANI, H., AND MONCASTER, T. Silo: Predictable message latency in the cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 435–448.

[36] NAGARAJ, K., BHARADIA, D., MAO, H., CHINCHALI, S., ALIZADEH, M., AND KATTI, S. Numfabric: Fast and flexible bandwidth allocation in datacenters. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference* (New York, NY, USA, 2016), SIGCOMM '16, ACM, pp. 188–201.

[37] CRONKITE-RATCLIFF, B., BERGMAN, A., VARGAFTIK, S., RAVI, M., MCKEOWN, N., ABRAHAM, I., AND KESLASSY, I. Virtualized congestion control. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference* (New York, NY, USA, 2016), SIGCOMM '16, ACM, pp. 230–243.

[38] 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 Conference on ACM SIGCOMM 2016 Conference* (New York, NY, USA, 2016), SIGCOMM '16, ACM, pp. 244–257.

Unobtrusive Deferred Update Stabilization for Efficient Geo-Replication

Chathuri Gunawardhana[1], *Manuel Bravo*[1,2] *and Luís Rodrigues*[1]
[1]INESC-ID, Instituto Superior Técnico, Universidade de Lisboa [2]Université Catholique de Louvain, Belgium

Abstract

In this paper, we propose a novel approach to manage the throughput vs visibility latency tradeoff that emerges when enforcing causal consistency in geo-replicated systems. Our approach consists in allowing full concurrency when processing local updates and using a deferred *local* serialisation procedure before shipping updates to remote datacenters. This strategy allows to implement inexpensive mechanisms to ensure system consistency requirements while avoiding intrusive effects on update operations, a major performance limitation of previous systems. We have implemented our approach as a variant of Riak KV. Our evaluation shows that we outperform sequencer-based approaches by almost an order of magnitude in the maximum achievable throughput. Furthermore, unlike previous sequencer-free solutions, our approach reaches nearly optimal remote update visibility latencies without limiting throughput.

1 Introduction

Geo-replication is a requirement for modern internet-based services in order to improve user-perceived latency. Unfortunately, due to the long network delays among sites, synchronous replication is prohibitively slow for most practical purposes. Therefore, many systems resort to weaker consistency semantics that permit some form of asynchronous replication strategy.

Among the many consistency guarantees that allow for asynchronous replication [15], causal consistency [9] has been identified as the strongest consistency model that an always-available system can implement [14, 37], becoming of practical relevance in geo-replicated settings. In fact, causal consistency is key in many geo-replicated storage systems offering from weak [38, 35, 12, 44] to strong consistency guarantees [41, 34, 17].

Unfortunately, implementing causal consistency is costly due to the computation, communication, and storage overhead caused by metadata management [19, 27, 16]. A common solution to reduce this cost consists in compressing metadata by serializing sources of concurrency, which unavoidably creates *false dependencies* among concurrent events, increasing *visibility latencies* (time interval between the instant in which an update is installed in its origin datacenter and when it becomes visible in remote datacenters).

To safely compress metadata, designers of causally consistent systems rely either on: (i) centralized *sequencers* (commonly one per datacenter) [44, 12]; or (ii) *global stabilization* procedures [24, 10] (executed across datacenters). The former has the advantage of making trivial—and therefore inexpensive—the dependency checking procedures at the cost of severely limiting concurrency, as sequencers operate in the critical path of clients. On the contrary, the latter avoids centralized synchronization points at the cost of periodically running a global stabilization procedure in the background. The cost of this procedure has pushed some systems to over-compress metadata to avoid impairing throughput, with a significant penalty on the visibility latencies [24].

In this paper, we propose, implement, and evaluate a novel approach to address the metadata size versus visibility latency tradeoff. Our approach has some similarities with systems that rely on global stabilization but also significant differences. As with [24, 10], we let local updates proceed without any a priori synchronization. However, unlike previous systems, we totally order all updates, in a manner consistent with causality, before shipping them to remote datacenters. As a result, expensive global stabilization is avoided, as it is trivial for a datacenter to check whether all updates subsumed in the timestamps piggybacked by remote updates have been locally applied (similarly to sequencer-based solutions).

We have implemented our approach as a variant of the open source version of Riak [6]. We have augmented Riak with Eunomia[1], a service that totally orders all lo-

[1]Greek goddess of law, her name can be translated as "good order".

cal updates, before shipping them. Our results show that Riak+Eunomia outperforms sequencer-based systems by almost an order of magnitude while serving significantly better quality-of-service to clients compared with systems based on global stabilization procedures.

In summary, the contributions of this paper are: i) The introduction of Eunomia, a new service for unobtrusively ordering updates (§3); ii) A fault tolerant version of Eunomia (§3.3); iii) An experimental comparison of the maximum load that traditional sequencers and Eunomia can handle, and their potential bottlenecks (§7.1); iv) The Integration of Eunomia into an always-available geo-replicated data store (§4) and its performance comparison to state-of-the-art solutions (§7.2).

2 Motivation and Goals

We start by motivating our work with a simple experiment, showing that: (i) the major throughput impairment of sequencer-based solutions is the fact that they operate in the critical path of clients; and (ii) global stabilization procedures are expensive in practice, forcing designers to favour either throughput or visibility latencies.

Figure 1 plots the throughput penalty and visibility latency overhead introduced by state-of-the-art causally consistent solutions. Results are normalized against an eventually consistent system, which adds no overhead due to consistency management. We vary from 1*ms* to 100*ms* the interval between global stabilization computations to better understand the cost and the consequences of such mechanism. Our deployment consists of 3 datacenters. The round-trip-times across datacenters are 80*ms* between datacenter 1 (dc_1) and both dc_2 and dc_3; and 160*ms* between dc_2 and dc_3. In the figure (left plot), latencies refer to the (90th percentile) delays incurred by each system at dc_2 for updates originating at dc_1. We compare the performance of 4 systems, namely *S-Seq*, *A-Seq*, *GentleRain* and *Cure*. For each solution, we deploy as many clients as possible (not necessarily the same amount for each experiment) without saturating the system.

S-Seq is a system that relies on a sequencer per datacenter to compress metadata; it uses a vector with an entry per datacenter to track causality, as in [12, 44]. *A-Seq* is an asynchronous (bogus) variant of *S-Seq*, that contacts the sequencer in parallel with applying the update. *A-Seq* does the same total amount of work as *S-Seq* and, although it fails to capture causality, it serves to reason about the potential benefits of removing sequencers from client's critical operational path. *GentleRain* [24] and *Cure* [10] are well known solutions that rely on global stabilization. The former favours throughput, over-compressing metadata into a single scalar; the latter favours visibility latencies, compressing metadata into a vector with an entry per datacenter.

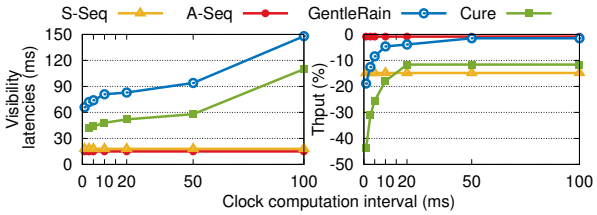


Figure 1: Update visibility latency vs throughput tradeoff.

The results confirm that the costs inherent to global stabilization force designers to choose between optimizing throughput and visibility latencies. As Figure 1 shows, Cure offers lower visibility latencies than GentleRain (as causality is more precisely tracked) at the cost of penalizing throughput. GentleRain does the opposite tradeoff favouring throughput. Cure can tune this tradeoff by choosing longer intervals among global stabilization occurrences. Nevertheless, even with long intervals (100*ms*), Cure still significantly degrades system throughput by 11.6%. Interestingly, results also show that by removing the sequencer from client's critical operational path, sequencer-based approaches could potentially pick a better spot in the tradeoff space, by providing throughput and visibility latencies comparable to GentleRain and Cure respectively, with almost no performance overhead when compared to the baseline. Note that in the above experiment, sequencers are not saturated; therefore, the throughout penalty (14.8%) is exclusively caused by the synchronous communication between the sequencer and the partitions at every client update operation. Later, in §7.1, we experimentally measure the maximum load that sequencers can handle before getting saturated.

From these results, it is possible to get the following insight: in order to alleviate the tension between throughput and visibility latencies, one has to (i) avoid global stabilization, and (ii) rely on an abstraction similar to sequencers that allows for trivial—therefore inexpensive—dependency checking procedures, while removing its operation from the client's critical path. Our goal was then to design Eunomia, a system with such characteristics.

3 Eunomia: Unobtrusive Ordering

In this section, we present the design and rationale underlying Eunomia, a new service conceived to replace sequencers as building blocks in weakly consistent geo-replicated storage systems. Unlike traditional sequencers, Eunomia lets local client operations to execute without synchronous coordination, an essential characteristic to avoid limiting concurrency and increasing operation latencies. Then, in the background, Eunomia establishes a serialization of all updates occurring in the local datacenter in an order consistent with causality, based

N	Number of partitions
$Clock_c$	Client c clock
p_n	Partition n
$Clock_n$	Current physical time at p_n
Ops	Set of unstable operations at Eunomia
$PartitionTime$	Vector with an entry per partition at Eunomia
$u_j.ts$	Timestamp assigned to update u_j

Table 1: Notation used in the protocol description.

Algorithm 1 Operations at client c

```
1:  function READ(Key)
2:      send READ(Key) to RESPONSIBLE(Key)
3:      receive ⟨Value, Ts⟩ from RESPONSIBLE(Key)
4:      Clock_c ← MAX(Clock_c, Ts)
5:      return Value
6:  function UPDATE(Key, Value)
7:      send UPDATE(Key, Value, Clock_c) to RESPONSIBLE(Key)
8:      receive Ts from RESPONSIBLE(Key)
9:      Clock_c ← Ts
10:     return ok
```

on timestamps generated locally by the individual servers that compose the datacenter. We refer to this process as *site stabilization procedure*. Thus, Eunomia is capable of abstracting the internal complexity of a multi-server datacenter without limiting the concurrency. Eunomia can be used to improve any existing sequencer-based solution to enforce causal consistency across geo-locations [38, 44, 12], as shown in §4.

3.1 Eunomia Into Play

In order to convey how Eunomia works, we start by presenting the protocol used to support the interaction between Eunomia and the machines that constitute a datacenter. In the exposition, we assume that the object-space is divided into *N partitions* distributed among datacenter machines. Updates to objects belonging to the same partition are serialized by the native update protocol. To simplify the presentation, our pseudocode assumes FIFO links among partitions and Eunomia. Later, in §3.3, we eliminate this assumption, making its implementation explicit. Table 1 provides a summary of the notation used in the protocols.

Eunomia assumes that each individual partition can assign a timestamp to each update without engaging in synchronous coordination with other partitions, or with Eunomia. We will explain below how this can be easily achieved. These timestamps must satisfy two properties.

Property 1. If an update u_j causally depends on a second update u_i, then the timestamp assigned to u_j ($u_j.ts$) is strictly greater than $u_i.ts$.

Property 2. For two updates u_i and u_j received by Eunomia from partition p_n, if u_i is received before u_j then $u_j.ts$ is strictly greater than $u_i.ts$.

These two properties imply that updates are causally ordered across all partitions and that once Eunomia receives an update coming from a partition p_n, no update with smaller timestamp will be ever received from p_n. In order to ensure these properties, clients play a fundamental role. A client c maintains a local variable, $Clock_c$, that stores the largest timestamp seen during its session. This clock value captures the client's causal dependencies and it is included in every update request. As described below, partitions compute update timestamps taking into account the value of client clocks.

Algorithm 2 Operations at partition p_n

```
1:  function READ(Key)
2:      ⟨Value, Ts⟩ ← KV_GET(Key)
3:      return ⟨Value, Ts⟩
4:  function UPDATE(Key, Value, Clock_c)
5:      MaxTs_n ← MAX(Clock_n, Clock_c + 1, MaxTs_n + 1)
6:      KV_PUT(Key, ⟨Value, MaxTs_n⟩)
7:      u_j ← ⟨Key, Value, MaxTs_n, p_n⟩
8:      send ADD_OP(u_j) to Eunomia
9:      return MaxTs_n
10: function HEARTBEAT                          ▷ Every Δ time
11:     if Clock_n ≥ MaxTs_n + Δ then
12:         send HEARTBEAT(p_n, Clock_n) to Eunomia
```

The protocol assumes that each partition p_n is equipped with a physical clock. Clocks are loosely synchronized by a time synchronization protocol such as NTP [5]. The correctness of the protocol does not depend on the clock synchronization precision and can tolerate clock drifts. However, as discussed later, large clock drifts could have a negative impact on the protocol performance (in particular, on how fast the datacenter can ship updates to remote datacenters). To avoid this limitation, our protocol uses hybrid clocks [30], which have been shown to overcome some of the limitations of simply using physical time.

We now describe how events are handled by clients, partitions and Eunomia (Algs. 1, 2, and 3 respectively).

Read. A client c sends a read request on item Key to the responsible partition p_n (Alg. 1, line 2). When p_n receives the request, it fetches the $Value$ and the timestamp Ts that is locally stored for Key and returns both to the client. Ts is the timestamp assigned by p_n to the update operation that generated the current version. After receiving the pair $\langle Value, Ts\rangle$, the client computes the maximum between $Clock_c$ and Ts (Alg. 1, line 4) to include the read operation in its causal history.

Update. A client c sends an update request operation to the responsible partition p_n of the object being updated. Apart from the Key and $Value$, the request includes the client's clock $Clock_c$ (Alg. 1, line 7). When p_n receives the request, it first computes the timestamp of the new update (Alg. 2, line 5). This is computed by taking the maximum between $Clock_n$ (physical time), the maximum timestamp ever used by p_n ($MaxTs_n$) plus one and $Clock_c$ (client's clock) plus one. This ensures that

Algorithm 3 Operations at Eunomia

```
1:  function ADD_OP(u_j)
2:      Ops ← Ops ∪ u_j
3:      ⟨Key, Value, Ts, p_n⟩ ← u_j
4:      PartitionTime[p_n] ← Ts
5:  function HEARTBEAT(p_n, Ts)
6:      PartitionTime[p_n] ← Ts
7:  function PROCESS_STABLE                          ▷ Every θ time
8:      StableTime ← MIN(PartitionTime)
9:      StableOps ← FIND_STABLE(Ops, StableTime)
10:     PROCESS(StableOps)
11:     Ops ← Ops \ StableOps
```

the timestamp is greater than both $Clock_c$ and any other update timestamped by p_n. Then, p_n stores the *Value* and the recently computed timestamp in the local key-value store and asynchronously sends the operation to the Eunomia service. Finally, p_n returns update's timestamp to the client who updates $Clock_c$ with it, since it is guaranteed to be greater than its current one.

Timestamp Stability. When Eunomia receives an operation from a given partition, it adds it to the set of non-stable operations Ops and updates the p_n entry in the *PartitionTime* vector with operation's timestamp (Alg. 3, lines 2–4). A timestamp Ts is *stable* at Eunomia when one is sure that no update with lower timestamp will be received from any partition (i.e., when Eunomia is aware of all updates with timestamp Ts or smaller). Periodically, Eunomia computes the value of the maximum stable timestamp (*StableTime*), which is computed as the minimum of the *PartitionTime* vector (Alg. 3, line 8). Property 2 implies that no partition will ever timestamp an update with an equal or smaller timestamp than *StableTime*. Thus, Eunomia can confidently serialize all operations tagged with a timestamp smaller than or equal to *StableTime* (Alg. 3, line 9). Eunomia can serialize them in timestamp order, which is consistent to causality (Property 1), and then send them to other geo-locations (Alg. 3, line 10). Note that non-causally related updates coming from different partitions may have been timestamped with the same value. In this case, operations are concurrent and Eunomia can process them in any order.

Heartbeats. If a partition p_n does not receive an update for a fixed period of time, it will send a heartbeat including its current time to Eunomia (Alg. 2, lines 10–12). Thus, even if a partition p_n receives updates at a slower pace than others, it will not slow down the processing of other partitions updates at Eunomia. When Eunomia receives a heartbeat from p_n, it simply updates its entry in the *PartitionTime* vector (Alg. 3, line 6).

Hybrid Clocks. Our protocol combines logical and physical time. Although Eunomia could simply use logical clocks and still be correct, the rate at which clocks from different partitions progress would depend on the rate in which partitions receive update requests. This may cause Eunomia to process local updates in a slower pace and thus increase remote visibility latencies, as the stable time is set to the smallest timestamp received among all partitions. Differently, physical clocks naturally progress at similar rates independently of the workload characterization. This fact—previously exploited by [24, 10]—makes stabilization procedures resilient to skewed load distribution. Unfortunately, physical clocks do not progress exactly at the same rate, forcing protocols to wait for clocks to catch up in some situations in order to ensure correctness [23, 24, 10, 25]. The logical part of the hybrid clock makes the protocol resilient to clock skew by avoiding artificial delays due to clock synchronization uncertainties [30]. Briefly, if a partition p_n receives an update request with $Clock_c > Clock_n$, instead of waiting until $Clock_n > Clock_c$ to ensure correctness, the logical part of the hybrid clock ($MaxTs_n$) is moved forward. Then, when a partition p_n receives an update from any client, if the physical part $Clock_n$ is still behind the logical ($MaxTs_n$), the update is tagged with $MaxTs_n + 1$ in order to ensure clock monotonicity and thus guarantee Property 2. The interested reader can find the correctness proof of the algorithm in [29].

3.2 Resilience to Stragglers

A straggler is a partition that, due to a transient lack of network or processing resources, experiences delays in contacting other system components. Naturally, stragglers do not affect only Eunomia, but affect any system that attempts to provide the same guaranties. Here, we discuss how Eunomia differs from other solutions when coping with stragglers (later in §7.2.3, we report on experiments with stragglers). We distinguish delays that affect the communication between distinct datacenters (*inter-dc stragglers*) and delays that affect the interaction of components inside the same datacenter (*intra-dc stragglers*). We expect the former to be more frequent than the latter [11, 26].

Inter-dc stragglers have a similar impact on every system, no matter it is sequencer-based or stabilization-based (Eunomia, GentleRain [24], Cure [10]). The reason is that inter-dc disturbances affect the transmission of the data and, therefore, delays the visibility of updates in a way that is orthogonal to the metadata scheme used.

Intra-dc stragglers are more interesting, because they affect different approaches in different ways. In a sequencer-based approach, the straggler experiences delays when contacting the sequencer, which happens before the update takes place. Therefore, intra-dc stragglers affect local clients (because sequencer operation is in client's critical path) but have no effect on the remote visibility of updates from healthy partitions. Conversely, in stabilization-based approaches, local clients

are shielded from the instability (because stabilization is performed in the background) but the remote visibility of updates from healthy partitions of the straggler's datacenter is affected (because only stable updates are propagated/applied and the contribution of all partitions is required to achieve stability). Although there is a trade-off, given that there is evidence that an increase in the user-perceived latency may translate into concrete revenue loss [40], we argue that stragglers may affect more sequencer-based approaches.

3.3 Fault-Tolerance

In the description above, for simplicity, we have described the Eunomia service as if implemented by a single non-replicated server. Naturally, as any other service in a datacenter, Eunomia must be made fault-tolerant. In fact, if Eunomia fails, the site stabilization procedure stops, and thus, local updates can no longer be propagated to other geo-locations. In order to avoid such limitation, we now propose a fault-tolerant version of Eunomia. Note that we disregard failures in datacenters, as the problem of making data services fault-tolerant has been widely studied and is orthogonal to our work.

In this new version, Eunomia is composed by a set of *Replicas*. Algorithm 4 shows the behaviour of a replica e_f of the fault-tolerant Eunomia service. We assume the initial set of Eunomia replicas is common knowledge: every replica knows every other replica and every partition knows the full set of replicas. Partitions send operations and heartbeats (Alg. 2, lines 8 and 12 respectively) to the whole set of Eunomia replicas. The correctness of the algorithm requires the communication between partitions and Eunomia replicas to satisfy the *prefix-property* [38]: an Eunomia replica r_f that holds an update u_j originating at p_n also holds any other update u_i originating at p_n such that $u_i.ts < u_j.ts$. This property can be ensured with inexpensive protocols that offer only *at-least-once delivery*. Stronger properties, such as inter-partition order or exactly-once delivery are not required to enforce the prefix-property. Our implementation achieves the prefix-property by having each partition to keep track of the latest timestamp acknowledged by each of the Eunomia replicas in a vector denoted as Ack_n. Thus, to each Eunomia replica e_f, a partition p_n sends not only the lastest update but the set of updates including all updates u_j such that $u_j.ts > Ack_n[f]$. Upon receiving a new batch of updates *Batch* (Alg. 4, lines 1–5), e_f process it—in timestamp order—filtering out those updates already seen, and updating both Ops_f and $PartitionTime_f$ accordingly with the timestamps of the unseen updates. After processing *Batch*, e_f acknowledges p_n including the greatest timestamp observed from updates originating at p_n ($PartitionTime_f[p_n]$). This algorithm is

Algorithm 4 Operations at Eunomia replica e_f

```
1:  function NEW_BATCH(Batch, p_n)
2:      for all u_j ∈ Batch, PartitionTime_f[p_n] < u_j.ts do
3:          PartitionTime_f[p_n] ← u_j.ts
4:          Ops_f ← Ops_f ∪ u_j
5:      send ACK(PartitionTime_f[p_n]) to p_n
6:  function PROCESS_STABLE                          ▷ Every θ time
7:      if Leader_f == e_f then
8:          StableTime ← MIN(PartitionTime_f)
9:          StableOps ← FIND_STABLE(Ops_f, StableTime)
10:         PROCESS(StableOps)
11:         Ops_f ← Ops_f \ StableOps
12:         send STABLE(StableTime) to Replicas_f \ {e_f}
13: function STABLE(StableTime)
14:     StableOps ← FIND_STABLE(Ops_f, StableTime)
15:     Ops_f ← Ops_f \ StableOps
16:     for all p_n ∈ PartitionTime_f do
17:         PartitionTime_f[p_n] ← MAX(PartitionTime_f[p_n], StableTime)
18: function NEW_LEADER(e_g)
19:     Leader_f ← e_g
```

resilient to message lost and unordered delivery. Nevertheless, it adds redundancy, as replicas may receive the same update multiple times. §5 proposes a set of optimizations that aim to reduce this overhead.

In addition, to avoid unnecessary redundancy when exchanging metadata among datacenters, a leader replica is elected to propagate this information. The existence of a unique leader is not required for the correctness of the algorithm; it is simply a mechanism to save network resources. Thus, any leader election protocol designed for asynchronous systems (such as Ω [20]) can be plugged into our implementation. A change in the leadership is notified to a replica e_f through the NEW_LEADER function (Alg. 4, line 19). The notion of a leader is used to optimize the service's operation as follows. When the PROCESS_STABLE event is triggered, only the leader replica computes the new stable time and processes stable operations (Alg. 4, lines 7–10). Then, after operations have been processed, the leader sends the recently computed *StableTime* to the remaining replicas (Alg. 4, line 12). When replica e_f receives the new stable time, it removes the operations already known to be stable from its pending set of operations, since it is certain that those operations have been already processed (Alg. 4, lines 14–15).

4 Supporting Geo-replication

In our previous protocol, we have shown how to unobtrusively timestamp local updates in a partial order consistent with causality. In this section, we complete the protocol with the necessary mechanisms to ensure that remote updates—coming from other datacenters—are made visible locally without violating causality. Our solution resembles protocols implemented by other causally consistent geo-replicated storage systems [12,

M	Number of datacenters
$VClock_c$	Client c vector (M entries)
p_n^m	Partition n at datacenter m
r_m	Receiver at datacenter m
$SiteTime_m$	Applied updates vector at r_m
$Queue_m$	Queues of pending updates at r_m
$u_j.vts$	Update u_j timestamp vector (M entries)

Table 2: Notation used in the geo-replicated protocol extension.

44]. We assume a total of M datacenters, each of them replicating the full set of objects. Each datacenter uses the Eunomia service and thus propagates local updates in a total order consistent to causal consistency.

Apart from the Eunomia service, each datacenter is extended with a *receiver*. This component coordinates the execution of remote updates. Thus, it receives remote updates coming from remote Eunomia services (as a result of PROCESS_STABLE), and forwards them to the local datacenter partitions when its causal dependencies are satisfied. Standard replication techniques [43, 33, 13, 39] can be employed to make receivers robust to failures, as otherwise they represent a single point of failure.

In order to simplify the presentation, our pseudocode assumes FIFO links between each Eunomia service and the receivers. Nevertheless, this assumption can be easily dropped if the Eunomia service includes on every message send to a receiver, no only the latest update but all previous updates that have not been acknowledge (by the receiver) yet. This mechanism, which is similar to the one described in §3.3, preserves the prefix-property, and therefore tolerates message lost and unordered delivery.

We now explain how the metadata is enriched and the changes we need to apply to our previous algorithms. Table 2 summarizes the notation used in this section.

Updates are now tagged with a vector with an entry per datacenter, capturing inter-datacenter dependencies. The client clock is consequently also extended to a vector ($VClock_c$). We could easily adapt our protocols to use a single scalar, as in [24]. Nevertheless, vector clocks make a more efficient tracking of causal dependencies introducing no false dependencies across datacenters, which reduces the update visibility latency, at the cost of slightly increasing the storage and computation overhead. This overhead, unlike in [10], is negligible in our protocol as Eunomia allows for trivial dependency checking procedures. Note that the lower-bound update visibility latency for a system relying on vector clocks is the latency between the originator of the update and the remote datacenter, while with a single scalar it is the latency to the farthest datacenter.

Update. When a client c issues an update operation, it piggybacks its $VClock_c$ summarizing both local and remote dependencies. A partition p_n computes u_j vector timestamp ($u_j.vts$) as follows. First, the local entry of the vector $u_j.vts[m]$ is computed as the maximum between $Clock_n$, $MaxTs_n + 1$ and $VClock_c[m] + 1$, similarly to Algorithm 2, line 5. This permits Eunomia to still be able to causally order local updates based on $u_j.vts[m]$. Second, the remaining entries (remote datacenter entries) are assigned to their sibling entries in $VClock_c$. When the operation is completed, p_n returns $u_j.vts$ to the client who can directly substitutes its $VClock_c$ since $u_j.vts$ is known to be strictly greater than $VClock_c$.

Algorithm 5 Operations at r_m

```
1:  function NEW_UPDATE(u_j, k)
2:      Queue_m[k] ← [Queue_m[k]|u_j]                          ▷ add to tail
3:  function CHECK_PENDING                                     ▷ Every ρ time
4:      ⟨Queue_m, SiteTime_m⟩ ← FLUSH(1, Queue_m, SiteTime_m)
5:  function FLUSH(k, Queue_m, SiteTime_m)
6:      if k > M then
7:          return ⟨Queue_m, SiteTime_m⟩
8:      else if k = m then
9:          FLUSH(k + 1, Queue_m, SiteTime_m)
10:     else
11:         u_j ←HEAD(Queue_m[k])
12:         if ∀d ∈ M \ {m,k}, SiteTime_m[d] ≥ u_j.vts[d] then
13:             p_n^m ← RESPONSIBLE(u_j.key)
14:             send APPLY(u_j) to p_n^m
15:             receive ok from p_n^m
16:             SiteTime_m[k] ← u_j.vts[k]
17:             POP(Queue_m[k])
18:             FLUSH(1, Queue_m, SiteTime_m)
19:         else
20:             FLUSH(k + 1, Queue_m, SiteTime_m)
```

Read. Read operations execute as in Algorithms 1 and 2. The only difference is that the returned timestamp is a vector instead of a scalar. Thus, in order to update $VClock_c$, a client c applies the MAX operation per entry.

Update Propagation. The site stabilization procedure proceeds as before, totally ordering local updates based on the local entry of their vector timestamp ($u.vts[m]$). Eunomia propagates local updates to remote datacenters in $u.vts[m]$ order. Each update piggybacks its $u.vts$.

Remote Update Visibility. Algorithm 5 details receivers' operation. A receiver r_m maintains two important pieces of state: a queue of pending updates per remote datacenter ($Queue_m[k]$), and a vector with an entry per remote datacenter ($SiteTime_m$) indicating the latest update operation locally applied from each of the remote datacenters. When r_m receives a remote update u_j coming from datacenter k, it simply adds it to its corresponding queue. Periodically, r_m triggers the CHECK_PENDING function (Algorithm 5 lines 4 and 18). This function ensures, by means of the tail recursive FLUSH function, that no pending operation is left unexecuted. Two conditions have to be satisfied before sending an update u_j to the local partitions: *(i)* all previously received updates coming from k have already been applied locally; and *(ii)* u_j dependencies, which are subsumed in $u_j.vts$, are visible locally. Both conditions are trivially checked by relying on the information subsumed in $Queue_m$ and $SiteTime_m$. When a pending operation

u_j originating at k is applied, both $Queue_m[k]$ and $SiteTime_m[k]$ are updated consequently.

5 Optimizations

We propose a set of optimizations that aim at enabling Eunomia to handle even heavier loads.

Communication Patterns. Eunomia constantly receives operations and heartbeats from partitions. This is an all-to-one communication schema and, if the number of partitions is large, it may not scale in practice. In order to overcome this problem and efficiently manage a large number of partitions, two simple techniques have been used: *(i)* build a propagation tree among partition servers; and *(ii)* batch operations at partitions, and propagate them to Eunomia only periodically. Both techniques are able to reduce the number of messages received by Eunomia per unit of time at the cost of a slight increase in the stabilization time.

Separation of Data and Metadata. In the protocols described before, partitions send updates (including the update value) to the Eunomia service, which is responsible for eventually propagating them to remote datacenters. This can limit the maximum load that Eunomia can handle and become a bottleneck due to the potentially large amount of data that has to be handled. In order to overcome this limitation, we decouple data from metadata.

In our prototype, for each update operation, partitions generate a unique update identifier ($u.id$), composed of the local entry of the update vector timestamp ($u.vts[m]$) and the object identifier (*Key*). We avoid sending the value of the update to Eunomia. Instead, partitions only send the unique identifier $u.id$ together with the partition id (p_n^m). Eunomia is then only responsible for handling and propagating these lightweight identifiers, while the partitions itself are responsible for propagating (with no order delivery constraints) the update values together with $u.id$ to its sibling partitions in other datacenters. A receiver r_m proceeds as before, but a partition p_n^m can only install the remote operation once it has received both the data and the metadata. This technique slightly increases the computation overhead at partitions, but it allows Eunomia to handle a significantly heavier load independently of update payloads.

6 Implementation

The Eunomia service is approximately 200 lines of `C++` code[2]. We integrated it with a version of Riak KV [6], a very popular [3] weakly consistent datastore used by many companies offering cloud-based services including Uber [2], bet365 [2] and Rovio [7]. Its integration consisted of 100 lines of Erlang code. We expect that integrating Eunomia into other popular NoSQL datastores such as Cassandra [32] would require a comparable effort as these datastores are architecturally very similar.

Since Riak KV is implemented in Erlang, we first attempted to build Eunomia using the `Erlang/OTP` framework, but unfortunately we reached a bottleneck in our early experiments. Note that for Eunomia to work, we need to store a potentially large number of updates, coming from all logical partitions composing a datacenter, and periodically traverse them in timestamp order when a new stable time is computed. Inserting and traversing this (ordered) set of updates was limiting the maximum load that Eunomia could handle, as accessing an item in a list using the built-in `Erlang` data type requires linear time with the number of elements in the list. The `C++` version does not suffer from these limitations.

At its core, Eunomia is uses a *red-black tree* [28], a self-balancing binary search tree optimized for insertions and deletions, which guarantees logarithmic search, insert and delete cost, and linear in-order traversal cost, a critical operation for Eunomia. In our case, the *red-black tree* turned out to be more efficient than other self-balancing binary search trees such as AVL trees [8].

Furthermore, in order to fully explore the capacities of Eunomia, we have integrated Eunomia with a causally consistent geo-replicated datastore implementing the protocol presented in §3 and §4. Our prototype, namely EunomiaKV[3], is built as a variant of Riak KV [6], and includes the optimizations discussed in §5. Since the open source version of Riak KV does not support replication across Riak KV clusters, we have also augmented it with geo-replication support.

7 Evaluation

Our main goal with the evaluation is to show that Eunomia does not suffer from the limitations of the competing approaches. Therefore, we compare Eunomia both with approaches based on sequencers and based on global stabilization. We recall that the main disadvantage of sequencers is to throttle throughput, because they operate in the critical path of local clients. Therefore, we aim at showing that Eunomia does not compromise the intra-datacenter concurrency and can reach higher throughput that sequencer-based approaches. Conversely, the expensiveness of the global stabilization approach forces designers to favour either throughput or remote update visibility latencies. Thus, we also aim at showing that

[2] Available at https://github.com/chathurilanchana/C-Stabilizer/tree/master/src

[3] Available at https://github.com/chathurilanchana/riak_kv/tree/causal-dev-multidc-nostat-nostraggler

Eunomia optimizes both.

Experimental Setup. The experimental test-bed used is a private cloud composed by a set of virtual machines deployed over 20 physical machines (8 cores and 40 GB of RAM) connected via a Gigabit switch. Each VM, which runs Ubuntu 14.04, and is equipped with 2 (virtual) cores, 10GB disk and 9GB of RAM memory; is allocated in a different physical machine. Before running each experiment, physical clocks are synchronized using the NTP protocol [5] through a near NTP server.

Workload Generator. Each client VM runs its own instance of a custom version of Basho Bench [1], a benchmarking tool. For each experiment, we deploy as many client instances as possible without overloading the system. Latencies across datacenters are emulated using `netem` [4], a Linux network emulator tool. The values used in operations are a fixed binary of 100 bytes. Our key-space is composed by 100k keys. The ratio of reads and updates is varied depending on the experiment. Before running the experiments, we populate the database. Each experiment runs for more than 6 minutes. In our results, the first and the last minute of each experiment is ignored to avoid experimental artifacts.

7.1 Eunomia Throughput

We report on a number of experiments that aim at: *(i)* measuring the maximum load that our efficient implementation of Eunomia can handle, varying the number of partitions connected to it; and *(ii)* assessing how replication and failures affect Eunomia's performance.

For comparison, these experiments also compute the throughput upper-bound of a traditional sequencer. Our implementation of a sequencer mimics traditional implementations [44, 12]. In every update operation, datacenter partitions synchronously request a monotonically increasing number to the sequencer before returning to the client. We have also implemented a fault-tolerant version of the sequencer based on chain replication [43]: Replicas are organized in a chain. Partitions send requests to the head of the chain. Requests traverse the chain up to the tail. When the tail receives a request, it replies back to the partition, which returns to the client.

In order to stretch as much as possible the implementation, circumventing potential bottlenecks in the system, we directly connect clients to Eunomia, bypassing the data store. Thus, each client acts as a partition in a multi-server datacenter. This allowed us to emulate very large datacenters, with much more servers than the ones that were at our disposal for these experiments, and overload Eunomia in a way that would be otherwise impossible with our testbed.

Throughput Upper-Bound. We first compare the non fault-tolerant version of the Eunomia against a non fault-tolerant implementation of a sequencer. In these experiments, partitions batch updates and only send them to Eunomia after 1ms.

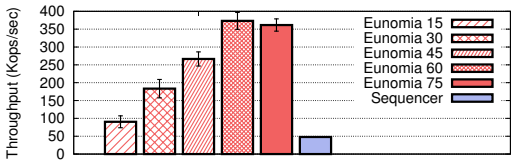


Figure 2: Maximum throughput achieved by Eunomia and an implementation of a sequencer. We vary the number of partitions that propagate operations to Eunomia.

Figure 2 plots the maximum throughput achieved by both services. As results show, Eunomia maximum throughput is reached when having 60 partitions issuing operations eagerly (with zero waiting time between operations). We observe that Eunomia is able to handle almost an order of magnitude more operations per second than a sequencer (more precisely, 7.7 times more operations, exceeding 370kops while the sequencer is saturated at 48kops). Considering that according to our experiments, a single machine in a Riak cluster is able to handle approximately 3kops per second, results confirm that sequencers limit intra-datacenter concurrency and can easily become a bottleneck for medium size clusters (i.e, for clusters above 150 machines, the sequencer would be the limiting factor of system performance), even assuming a read dominant (9:1) workload, a common workload for internet-based services. Nevertheless, under the same workload assumptions, more than a thousand machines could be used before saturating Eunomia.

Another advantage of Eunomia in comparison to sequencers is that batching is not in client's critical path. Thus, Eunomia's throughput can be further stretched by increasing the batching time (while slightly increasing the remote update visibility latency). Such stretching cannot be easily achieved with sequencers, as any attempt to batch requests at the sequencer blocks clients.

A final conclusion can be drawn from this experiment: Eunomia maximum capacity does not significantly varies with the number of partitions. Although we hit the maximum load with 60 partitions, we run an extra experiment increasing the number to 75 to see if this negatively impacts Eunomia's performance and we observed a very similar throughput. The reason is that the bottleneck of our Eunomia implementation is the propagation to other geo-locations rather than the handling of operations. This confirms that the use of a red-black self-balancing search tree was an appropriate design choice.

Fault-Tolerance Overhead. In the following experiments we measure the overhead introduced by the fault-tolerant version of Eunomia. Figure 3 compares the maximum throughput achievable by Eunomia when increas-

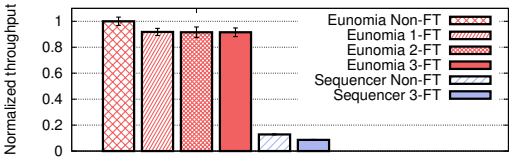


Figure 3: Maximum throughput achieved by a fault-tolerant version of Eunomia and sequencers. Non-FT denotes non fault-tolerant versions while 1-, 2-, and 3-FT denote fault-tolerant versions with 1, 2, and 3 replicas

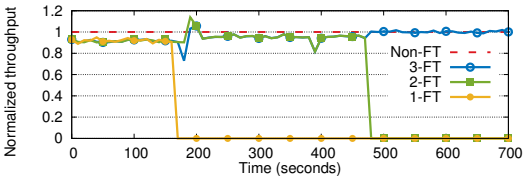


Figure 4: Impact of failures in Eunomia.

ing the number of replicas up to three. For completeness, the plot also includes the throughput for a non fault-tolerant sequencer and its fault-tolerant version with a chain of three replicas. We normalized the throughput against the non fault-tolerant version of Eunomia. As results show, the fault-tolerant version of Eunomia only adds a small overhead (roughly 9% penalty) independently on the number of replicas. We expect this overhead to increase as the number of replicas increases, but we consider three replicas to be a realistic number. On the other hand, adding fault-tolerance to the sequencer version adds a penalty of almost 33%, thus being more expensive proportionally. The reason for this difference is that Eunomia replicas do not need to coordinate as their results are independent of relative order of inputs, while sequencer replicas need to coordinate to avoid providing inconsistent sequence numbers.

Impact of Failures. Finally, we experiment injecting failures into Eunomia. Figure 4 plots the results normalized against the non fault-tolerant Eunomia (Non-FT line). We compare Eunomia with one, two, and three replicas. As the figure shows, at the beginning of the experiment, all three versions produce similar throughput (confirming Figure 3 results). After 160 seconds, we crash one replica. As expected, the throughput of 1-FT drops to zero since no more replicas are available. The rest of the versions (2-FT and 3-FT), after a short period of fluctuation, slightly increase their throughput up to 95% of the Non-FT version throughput. Finally, after 210 more seconds (at 470), we crash a second replica. Again, the 2-FT as expected drops its throughput to zero. The 3-FT version, this time almost without fluctuations, is capable of achieving the maximum throughput in few seconds. These results demonstrate that failures have negligible impact in Eunomia. Note that sometimes the multi-replica version go beyond the Non-FT line because the Non-FT line is drawn by computing the average.

7.2 Experiments with Geo-Replication

We now report on a set of experiments offering evidence that a causally consistent geo-replicated datastore built using Eunomia is capable of providing higher throughput and better quality-of-service than previous solutions that avoid the use of local sequencers.

For this purpose, we have implemented GentleRain [24] and a variation of it that uses vector clocks instead of a single scalar to enforce causal consistency across geo-locations. The latter resembles the causally consistency protocol implemented by Cure [10]. Both approaches are sequencer-free that rely on a global stabilization procedure in order to apply operations in remote locations consistently with causality. For this, sibling partitions across datacenters have to periodically send heartbeats, and each partition within a datacenter has to periodically compute its local-datacenter stable time. In our experiments, we set the time interval of this events to 10ms and 5ms respectively unless otherwise specified. These values are in consonance to the ones used by the authors of these works. For a fair comparison, both approaches are implemented using the EunomiaKV's codebase and thus integrated with Riak KV.

In most of our experiments, we deploy 3 datacenters, each of them composed of 8 logical partitions balanced across 3 servers. The emulated round-trip-times across datacenters are 80ms between dc_1 and both dc_2 and dc_3, and 160ms between dc_2 and dc_3. These latencies are approximately the round-trip-times between Virginia, Oregon and Ireland regions of Amazon EC2.

7.2.1 Throughput

In the following experiments, we measure the throughput provided by EunomiaKV, GentleRain, Cure, and an eventually consistent multi-cluster version of Riak KV. Note that the latter does not enforce causality, and thus partitions install remote updates as soon as they are received. Therefore, the comparison of EunomiaKV with Riak KV allows to assess the overhead that enforcing causal consistency adds when using our approach. As discussed below, this overhead is very small.

We experiment with both uniform and power-law key distributions, denoted with U and P respectively in Figure 5. For each of them, we vary the read:write ratio (99:1, 90:10, 75:25 and 50:50). These ratios are representative of real large internet-based services workloads. As shown by Figure 5, the throughput of all solutions decreases as we increase the percentage of updates. Nevertheless, EunomiaKV always provides a comparable throughput to eventual consistency. Precisely, on average, EunomiaKV only drops 4.7% of throughput, being extremely close in read intensive workloads (1% drop). Differently, GentleRain and Cure are al-

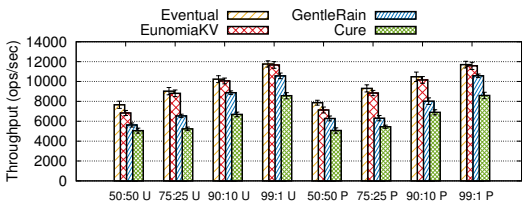


Figure 5: Throughput comparison between EunomiaKV and state-of-the-art sequencer-free solutions.

ways significantly below both eventual consistency (and EunomiaKV). This is due to the cost of the global stabilization procedure. Note that the throughput difference between GentleRain and Cure is caused by the overhead introduced by the metadata enrichment procedure of the latter (as discussed in §4). Based on our experiments, it is possible to conclude that the absolute number of updates per unit of time is the factor that has the largest impact in EunomiaKV (rather than key contention).

7.2.2 Remote Update Visibility

To compare the quality-of-service that can be provided by EunomiaKV, GentleRain, and Cure, we measure remote update visibility latencies. In EunomiaKV, we measure the time interval between the data arrival and the instant in which the update is executed at the responsible partition. Note that, for an update to be applied, a datacenter needs to have access to the metadata (in our case, provided by Eunomia) and check that all of its causal dependencies have also been previously applied locally. In our implementation, partitions ship updates immediately to remote datacenters. Therefore, we have observed that updates are always locally available to be applied by the time metadata indicates that its causal dependencies are already satisfied locally. Although other strategies could be used to ship the payload of the updates, this has a crucial advantage for the evaluation of Eunomia: under this deployment the update visibility latency is exclusively influenced by the performance of the metadata management strategy, including the stabilization delay incurred at the originating datacenter.

On the other hand, for GentleRain and Cure, we measure the time interval between the arrival of the remote operation to the partition and when the global stabilization procedure allows its visibility. Note that all values presented in the figures already factor-out the network latencies among datacenters (which are the same for all protocols); thus numbers capture only the artificial artifacts inherent to the different approaches.

Figure 6 (left plot) shows the cumulative distribution of the latency before updates originating at dc_1 become visible at dc_2. We observe that EunomiaKV offers, by far, the best remote update visibility latency. In fact, for almost 95% of remote updates, EunomiaKV only adds 15ms extra delay. On the other hand, with GentleRain

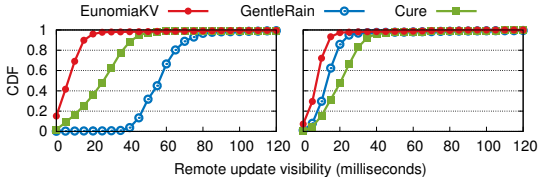


Figure 6: Left: from dc_1 to dc_2 (40ms trip-time). Right: from dc_2 to dc_3 (80ms trip-time).

and Cure the extra delay goes up to 80ms and 45ms respectively for the same amount of updates. Unsurprisingly, GentleRain extra delay is larger than Cure's because of the amount of false dependencies added when aggregating causal dependencies into a single scalar. In fact, GentleRain is not capable of making updates visible without adding 40ms of extra delay. Again, the scalar is the cause of this phenomenon since the minimum delay will not depend on the originator of the update but on the travel time to the furthest datacenter. This confirms the rationale presented in the discussion of §4.

Although both Cure and EunomiaKV rely on vector clocks for tracking causal dependencies, EunomiaKV is able to offer better remote update latencies because partitions are less overloaded since checking dependencies in EunomiaKV is trivial due to Eunomia. Note that in EunomiaKV, even 20% of remote updates are made visible without any extra delay, and thus reaching the optimal remote update visibility latency.

Finally, in order to isolate the impact of GentleRain's global stabilization procedure independently of the metadata size, we measure the remote update visibility latency at dc_3 for updates originating at dc_2. As one can observe in Figure 6 (right plot), GentleRain exhibits better remote update latencies than Cure but still worse than EunomiaKV. In this setting, vector clocks does not help reducing latencies. Thus, the gap between Cure and GentleRain is exclusively due to the storage and computational overhead caused by vector clocks. Furthermore, the fact that EunomiaKV still provides better latencies is, once again, an empirical evidence that global stabilization procedures are expensive in practice.

7.2.3 Impact of Stragglers

Finally, we assess the impact of stragglers in EunomiaKV and its competitors. Due to lack of space, and given that they provide no significant insight, we omit experimental results for inter-dc stragglers.

In these experiments, we use three datacenters (same setup of previous experiments) that run under optimal conditions during 1 minute. Then, during the second minute, we introduce a straggler. This is a partition of dc_3 that communicates abnormally with its local sequencer or Eunomia service. In Eunomia, instead of communicating every millisecond (as every other parti-

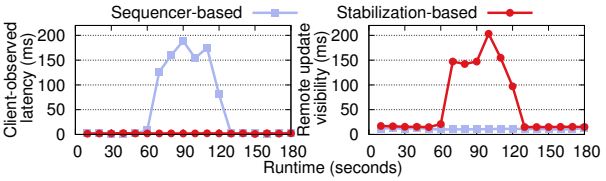


Figure 7: Client-observed latency (measured as the averaged latency observed by clients of the straggling datacenter dc_3) vs. remote update visibility latency (measured at dc_2 for updates originating at dc_3) tradeoff disclosed by intra-dc stragglers (one second straggling interval).

tion), the straggler contacts Eunomia less frequently. In the sequencer-based system, a similar delay (on average) is introduced when the straggler partition contacts the sequencer. We have experimented with three straggling intervals: 10, 100 and 1000ms, all exhibiting similar patterns. Figure 7 shows results for a 1 second straggling interval, as it is the most striking result. After the straggling period, the partition gets healed.

As expected (§3.2), intra-dc stragglers do not affect the remote visibility of updates in sequencer-based approaches but clients notice a significant increase in latency. In contrast, stabilization-based approaches are capable of shielding clients from stragglers and the cost of increasing the remote visibility of updates. Note that the stabilization-based results were obtained with Eunomia, but GentleRain and Cure exhibit a similar behaviour.

8 Related work

The support for causal consistency can already be found in early pioneer works in distributed systems, such as Bayou [38, 42], Lazy Replication [31], and the ISIS [18] toolkit. Recently, and tackling scalability challenges close to ours, multiple weakly consistent geo-replicated data stores implementing causal consistency across geo-locations have been proposed. We group them into two categories: *(i)* sequencer-based solutions [12, 44, 21]; *(ii)* and sequencer-free solutions [35, 22, 36, 24, 10].

Sequencer-based. These solutions rely on a sequencer per datacenter to enforce causal consistency. The sequencer totally orders local updates, in a causally consistent manner, and propagate them to remote locations. This design centralizes, thus simplifying, the implementation of causal consistency. Nevertheless, the use of synchronous sequencers limits the intra-datacenter concurrency, as demonstrated by our experiments. SwiftCloud [44] and ChainReaction [12] rely on a vector clock with an entry per datacenter to track causal dependencies, similarly to EunomiaKV. Practi [21], on the contrary, uses a single scalar and a sophisticated mechanism of invalidations. Similar to EunomiaKV, Practi separates the propagation of data and metadata. This and the concept of *imprecise* invalidations optimize Practi for partial replication, a setting that has not yet been explored in this work. We have shown that sequencers may get easily saturated for medium-size clusters, while Eunomia is able to handle much heavier loads (up to 7.7 times more).

Sequencer-free. There have been two major trends in this category: *(i)* solutions that rely on explicit dependency check messages [35, 22, 36]; and *(ii)* solutions based on global stabilization procedures [24, 10].

COPS [35] and Eiger [36] finely track dependencies for each individual data item allowing full concurrency within a datacenter. Updates are tagged with a list of dependencies. When a datacenter receives a remote update, it needs to explicitly check each dependency. This process is expensive and limits systems performance [24] due to the large amount of metadata managed. Orbe [22] aggregates dependencies belonging to the same logical partition into a scalar, only partially solving the problem.

Alternatives that use less metadata rely on a background global stabilization procedure [24, 10]. This procedure equips partitions with sufficient information to safely execute remote updates consistently with causality. Thus, these solutions manage to aggregate the metadata as sequencer-based solutions without relying on an actual sequencer. As our extensive evaluation has empirically demonstrated, global stabilization procedures are expensive in practice, forcing designers to favour either throughput [24] or remote visibility latency [10]. Our evaluation shows that EunomiaKV does not force designers to sacrifice any of the two, exhibiting significantly better throughput and remote visibility latencies than Cure and GentleRain respectively.

9 Conclusions

We have presented a novel approach for building causally consistent geo-replicated data stores. Our solution relies on Eunomia, a new service that abstracts the internal complexity of datacenters, a key feature to reduce the cost of causal consistency. Unlike sequencers, Eunomia does not limit the intra-datacenter concurrency by performing an unobtrusive ordering of updates. Our evaluation shows that Eunomia can handle very heavy loads without becoming a performance bottleneck (up to 7.7 times more operations per second than sequencers). Experiments also show that EunomiaKV (a causally consistent geo-replicated protocol that integrates Eunomia), unlike previous systems, permits optimizing both throughput and remote update visibility latency simultaneously. In fact, results have shown that EunomiaKV only adds a slight throughput overhead (4.7% on average) and exceptionally small artificial remote visibility delays when compared to an eventually consistent data store that makes no attempt to enforce causality.

Acknowledgments

We would like to thank our shepherd Chunqiang (CQ) Tang, Kuganesan Srijeyanthan, and anonymous reviewers for their comments and suggestions. This research has been supported in part by the Horizon 2020 project 732 505 LightKone, by the Erasmus Mundus Doctorate Programme under Grant Agreement No. 2012-0030, by the European Master in Distributed Computing (EMDC), and by FCT through projects PTDC/ EEI-SCR/ 1741/ 2014 (Abyss) and UID/ CEC/ 50021/ 2013.

References

[1] Basho Bench. http://github.com/basho/basho_bench.

[2] bet365. http://www.bet365.com/.

[3] Customers of Riak KV. http://basho.com/about/customers/.

[4] Netem. http://www.linuxfoundation.org/collaborate/workgroups/networking/netem.

[5] The network time protocol. http://www.ntp.org.

[6] Riak KV. https://github.com/basho/riak_kv.

[7] Rovio. http://www.rovio.com/.

[8] ADELSON-VELSKII, M., AND LANDIS, E. An algorithm for the organization of information. Tech. rep., DTIC Document, 1963.

[9] AHAMAD, M., NEIGER, G., BURNS, J. E., KOHLI, P., AND HUTTO, P. W. Causal memory: definitions, implementation, and programming. *Distributed Computing 9*, 1 (1995), 37–49.

[10] AKKOORATH, D., TOMSIC, A., BRAVO, M., LI, Z., CRAIN, T., BIENIUSA, A., PREGUIÇA, N., AND SHAPIRO, M. Cure: Strong semantics meets high availability and low latency. In *Proceedings of the International Conference on Distributed Computing Systems* (Osaka, Japan, 2016).

[11] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication* (Seattle, WA, USA, 2008), pp. 63–74.

[12] ALMEIDA, S., LEITÃO, J. A., AND RODRIGUES, L. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic, 2013).

[13] ALSBERG, P. A., AND DAY, J. D. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering* (San Francisco, CA, USA, 1976).

[14] ATTIYA, H., ELLEN, F., AND MORRISON, A. Limitations of highly-available eventually-consistent data stores. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (Donostia-San Sebastián, Spain, 2015).

[15] BAILIS, P., DAVIDSON, A., FEKETE, A., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Highly available transactions: Virtues and limitations. *Proc. VLDB Endow. 7*, 3 (Nov. 2013), 181–192.

[16] BAILIS, P., FEKETE, A., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the ACM Symposium on Cloud Computing* (San Jose, California, 2012).

[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 10th ACM European Conference on Computer Systems* (Bordeaux, France, 2015).

[18] BIRMAN, K., SCHIPER, A., AND STEPHENSON, P. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst. 9*, 3 (Aug. 1991).

[19] BRAVO, M., DIEGUES, N., ZENG, J., ROMANO, P., AND RODRIGUES, L. On the use of clocks to enforce consistency in the cloud. *IEEE Data Eng. Bull 38*, 1 (2015), 18–31.

[20] CHANDRA, T., HADZILACOS, V., AND TOUEG, S. The weakest failure detector for solving consensus. *J. ACM 43*, 4 (July 1996), 685–722.

[21] DAHLIN, M., GAO, L., NAYATE, A., VENKATARAMANA, A., YALAGANDULA, P., AND ZHENG, J. Practi replication. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation* (San Jose, CA, USA, 2006).

[22] DU, J., ELNIKETY, S., ROY, A., AND ZWAENEPOEL, W. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Clara, CA, USA, 2013).

[23] DU, J., ELNIKETY, S., AND ZWAENEPOEL, W. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Proceedings of the 32nd IEEE Symposium on Reliable Distributed Systems* (Braga, Portugal, 2013).

[24] DU, J., IORGULESCU, C., ROY, A., AND ZWAENEPOEL, W. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA, 2014).

[25] DU, J., SCIASCIA, D., ELNIKETY, S., ZWAENEPOEL, W., AND PEDONE, F. Clock-RSM: Low-latency inter-datacenter state machine replication using loosely synchronized physical clocks. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (Atlanta, Georgia USA, 2014).

[26] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication* (Barcelona, Spain, 2009), pp. 51–62.

[27] GUERRAOUI, R., PAVLOVIC, M., AND SEREDINSCHI, D.-A. Trade-offs in replicated systems. *Data Engineering* (2016), 14.

[28] GUIBAS, L. J., AND SEDGEWICK, R. A dichromatic framework for balanced trees. In *Proceedings of the 54th IEEE Annual Symposium on Foundations of Computer Science* (Ann Arbor, Michigan, USA, 1978), pp. 8–21.

[29] GUNAWARDHANA, C., BRAVO, M., AND RODRIGUES, L. Unobtrusive deferred update stabilization for efficient geo-replication. *arXiv:1702.01786 [cs.DC]* (Feb. 2017).

[30] KULKARNI, S. S., DEMIRBAS, M., MADAPPA, D., AVVA, B., AND LEONE, M. Logical physical clocks. In *Proceedings of the 18th International Conference on Principles of Distributed Systems* (Cortina d'Ampezzo, Italy, 2014).

[31] LADIN, R., LISKOV, B., SHRIRA, L., AND GHEMAWAT, S. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.* (1992).

[32] LAKSHMAN, A., AND MALIK, P. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[33] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 133–169.

[34] 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* (Hollywood, CA, USA, 2012), pp. 265–278.

[35] 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 23rd ACM Symposium on Operating Systems Principles* (Cascais, Portugal, 2011).

[36] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation* (Lombard, IL, USA, 2013).

[37] MAHAJAN, P., ALVISI, L., AND DAHLIN, M. Consistency, availability, and convergence. Tech. rep., University of Texas at Austin, 2011.

[38] PETERSEN, K., SPREITZER, M. J., TERRY, D. B., THEIMER, M. M., AND DEMERS, A. J. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (Saint Malo, France, 1997).

[39] SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv. 22*, 4 (Dec. 1990), 299–319.

[40] SCHURMAN, E., AND BRUTLAG, J. The user and business impact of server delays, additional bytes, and HTTP chunking in web search. In *Velocity Web Performance and Operations Conference* (San Jose, CA, USA, 2009).

[41] SOVRAN, Y., POWER, R., AGUILERA, M. K., AND LI, J. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (Cascais, Portugal, 2011).

[42] TERRY, D. B., DEMERS, A. J., PETERSEN, K., SPREITZER, M. J., THEIMER, M. M., AND WELCH, B. B. Session guarantees for weakly consistent replicated data. In *Proceedings of the 3rdInternational Conference on Parallel and Distributed Information Systems* (Austin, TX, USA, 1994).

[43] VAN RENESSE, R., AND SCHNEIDER, F. B. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th symposium on Operating systems design and implementation* (San Francisco, CA, USA, 2004).

[44] ZAWIRSKI, M., PREGUIÇA, N., DUARTE, S., BIENIUSA, A., BALEGAS, V., AND SHAPIRO, M. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the annual ACM/IFIP/USENIX Middleware conference* (Vancouver, Canada, 2015).

Don't cry over spilled records: Memory elasticity of data-parallel applications and its application to cluster scheduling

Călin Iorgulescu[*], Florin Dinu[*], Aunn Raza[‡], Wajih Ul Hassan[†], and Willy Zwaenepoel[*]

[*]EPFL [‡]NUST Pakistan [†]UIUC

Abstract

Understanding the performance of data-parallel workloads when resource-constrained has significant practical importance but unfortunately has received only limited attention. This paper identifies, quantifies and demonstrates memory elasticity, an intrinsic property of data-parallel tasks. Memory elasticity allows tasks to run with significantly less memory than they would ideally need while only paying a moderate performance penalty. For example, we find that given as little as 10% of ideal memory, PageRank and NutchIndexing Hadoop reducers become only 1.2x/1.75x and 1.08x slower. We show that memory elasticity is prevalent in the Hadoop, Spark, Tez and Flink frameworks. We also show that memory elasticity is predictable in nature by building simple models for Hadoop and extending them to Tez and Spark.

To demonstrate the potential benefits of leveraging memory elasticity, this paper further explores its application to cluster scheduling. In this setting, we observe that the resource vs. time trade-off enabled by memory elasticity becomes a task queuing time vs. task runtime trade-off. Tasks may complete faster when scheduled with less memory because their waiting time is reduced. We show that a scheduler can turn this task-level trade-off into improved job completion time and cluster-wide memory utilization. We have integrated memory elasticity into Apache YARN. We show gains of up to 60% in average job completion time on a 50-node Hadoop cluster. Extensive simulations show similar improvements over a large number of scenarios.

1 Introduction

The recent proliferation of data-parallel workloads [27, 10, 24] has made efficient resource management [22, 26, 7] a top priority in today's computing clusters. A popular approach is to better estimate workload resource needs to avoid resource wastage due to user-driven over-estimations [26, 12, 21]. Another is over-committing server resources to cope with the variability of workload resource usage [26, 7, 11]. Unfortunately, only a few efforts [12] have touched on the malleability of data-parallel workloads when resource-constrained. The study of malleability is complementary to solutions for over-estimations and variability. While the latter two attempt to accurately track the actual workload resource usage, the former is about allocating to applications fewer server resources than they would ideally need. A thorough understanding of the trade-offs involved in resource malleability is useful in many contexts ranging from improving cluster-wide resource efficiency and provisioning to reservation sizing in public clouds, disaster and failure recovery, and cluster scheduling.

The main contribution of this paper is identifying, quantifying and demonstrating memory elasticity, an intrinsic property of data-parallel workloads. We define memory elasticity as the property of a data-parallel task to execute with only a moderate performance penalty when memory-constrained. Memory elasticity pertains to tasks involved in data shuffling operations. Data shuffling is ubiquitous [6, 20, 28]. It is used by a large number of data-parallel applications across all data-parallel frameworks. Most tasks are involved in shuffling and show memory elasticity: mappers and reducers in MapReduce, joins and by-key transformations (reduce, sort, group) in Spark, and mappers, intermediate and final reducers in Tez.

Despite significant differences in the designs of popular data-parallel frameworks, shuffling operations across these frameworks share a common, tried-and-tested foundation in the use of merge-sort algorithms that may also use secondary storage [3]. The memory allocated to a task involved in shuffling has a part for shuffling and a part for execution. The best task runtime is obtained when the shuffle memory is sized such that all shuffle data fits in it. This allows the shuffle to perform an effi-

†‡ Work done while authors were interns at EPFL.

cient in-memory-only merge-sort. If the shuffle memory is insufficient, an external merge-sort algorithm is used.

The key insight behind memory elasticity is that under-sizing shuffle memory can lead to considerable reductions in task memory allocations at the expense of only moderate increases in task runtime. Two factors contribute to the sizeable memory reductions. First, shuffle memory is usually a very large portion of the task memory allocation (70% by default in Hadoop). Second, external merge-sort algorithms can run with very little memory because they can compensate by using secondary storage. A couple of factors also explain why the task runtime increases only moderately when shuffle memory is under-sized. First, a data-parallel task couples shuffling with CPU-intensive processing thus making far less relevant the performance gap between external and in-memory merge-sort. Second, disk accesses are efficient as the disk is accessed sequentially. Third, the performance of external merge-sort algorithms remains stable despite significant reductions in shuffle memory (a k-way merge is logarithmic in k).

Thus, memory elasticity presents an interesting resource vs. time trade-off. This paper quantifies this trade-off and its implications using extensive experimental studies. We find that memory elasticity is prevalent across the Hadoop, Spark, Tez and Flink frameworks and across several popular workloads. In all cases, the performance penalty of memory elasticity was moderate despite sizeable reductions in task memory allocations. Let M be the task memory allocation that minimizes task runtime by ensuring that all shuffle data fits in shuffle memory. Given as little as 10% of M, PageRank and NutchIndexing Hadoop reducers become only 1.22x/1.75x and 1.08x slower. For Hadoop mappers the largest encountered penalty is only 1.5x. For Spark, Tez and Flink the penalties were similar to Hadoop. Furthermore, we show the predictable nature of memory elasticity which is key to leveraging it in practice. We build simple models for Hadoop that can accurately describe the resource vs. time trade-off. With only small changes, the same models apply to Spark and Tez.

To demonstrate the potential benefits of leveraging memory elasticity, this paper further explores its application to cluster scheduling. Current clusters host concurrently a multitude of jobs each running a multitude of tasks. In this setting, we observe that the resource vs. time trade-off of memory elasticity becomes a task queueing time vs. task runtime trade-off. A task normally has to wait until enough memory becomes available for it but if it is willing to execute using less memory it might have to wait much less or not at all. Since the completion time of a task is the sum of waiting time plus runtime, a significant decrease in waiting time may outweigh an increase in runtime due to elasticity and overall lead to faster task completion times. We show that a scheduler can turn this task-level trade-off into improved job completion time and improved cluster-wide memory utilization by better packing tasks on nodes with respect to memory. Scheduling using memory elasticity is an NP-hard problem because it contains as a special case NP-hard variants of the RCPSP problem [8], a well-known problem in operations research. We propose a simple heuristic and show it can yield important benefits: the tasks in a job can leverage memory elasticity only if that does not lead to a degradation in job completion time.

We have integrated the concepts of memory elasticity into Apache YARN. On a 50-node Hadoop cluster, leveraging memory elasticity results in up to 60% improvement in average job completion time compared to stock YARN. Extensive simulations show similar improvements over a large number of scenarios.

2 Memory elasticity in real workloads

This section presents an extensive study of memory elasticity. We make three key points. First, memory elasticity is generally applicable to several frameworks and workloads. Our measurements put emphasis on Hadoop but also show that elasticity applies to Apache Spark, Tez and Flink. Second, memory elasticity costs little. The performance degradation due to using elasticity was moderate in all experiments. Third, elasticity has a predictable nature and thus can be readily modeled. We provide a model for Hadoop and with only simple changes apply it to Tez and a Spark Terasort job. We also detail the causes and implications of memory elasticity.

We use the term *spilling to disk* to refer to the usage of secondary storage by the external merge-sort algorithms. We call a task *under-sized* if its memory allocation is insufficient to avoid spilling to disk during shuffling. We call a task *well-sized* otherwise. We call *ideal memory* the minimum memory allocation that makes a task well-sized and *ideal runtime* the task runtime when allocated ideal memory. The term *penalty* refers to the performance penalty caused by memory elasticity in under-sized tasks.

2.1 Measurement methodology

For Hadoop we profiled 18 jobs across 8 different applications, most belonging to the popular HiBench big-data benchmarking suite [4]. The jobs range from graph processing (Connected Components, PageRank) to web-indexing (Nutch), machine learning (Bayesian Classification, Item-Based Recommender), database queries (TPC-DS) and simple jobs (WordCount, TeraSort). For Spark we profiled TeraSort and WordCount, for Tez we profiled WordCount and SortMerge Join and for Flink

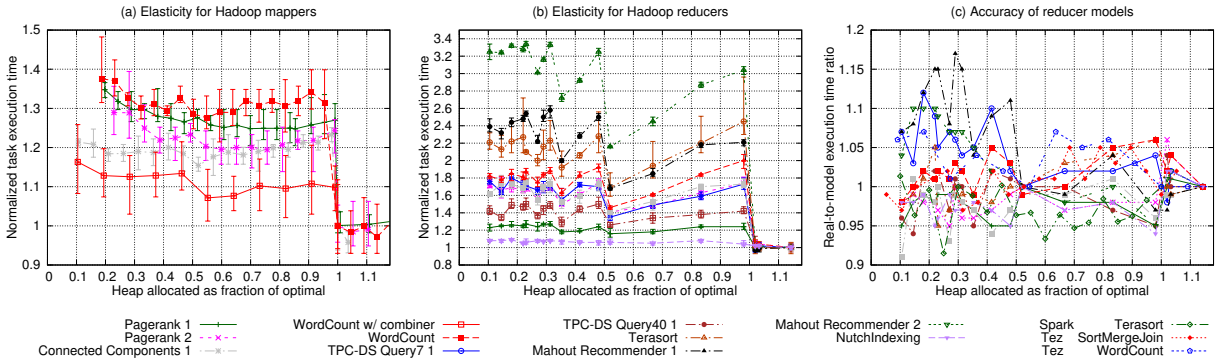


Figure 1: Memory elasticity profiles for Hadoop mappers (a) and reducers (b). Accuracy of our model (c). The reducers (b) exhibit a sawtooth-like pattern determined by the input-to-buffer ratio. Local minimum/maximum penalty values occur right before and after the points where this ratio is an integer (further detailed in §2.3).

we profiled WordCount. We used Hadoop 2.6.3, Spark 2.0.0, Tez 0.7.0 and Flink 1.0.2. However, the same behavior appears in Spark versions prior to 2.0.0 and Hadoop versions at least as old as 2.4.1 (June 2014).

For accurate profiling we made sure that the profiled task is not collocated with any other task. To measure the worst case penalties for under-sized tasks we ensure that disk I/O operations for spills actually go to the drive and not to the OS buffer cache. For this, we ran each task in a separate Linux cgroups container. We minimize the amount of buffer cache available to a task by setting the cgroups limits as close to the JVM heap size as possible. As an alternative, we also modified Hadoop to perform disk spills using direct I/O thus bypassing completely the OS buffer cache. The two approaches gave consistently similar results.

2.2 Memory elasticity for Hadoop mappers

Elasticity for mappers occurs on their output side. The key-value pairs output by map function calls are written to an in-memory buffer. If the mapper is well-sized then the buffer never fills up. In this case, when the mapper finishes processing its input, the buffer contents are written to disk into one sorted and partitioned file (one partition per reducer). If the mapper is under-sized, the buffer fills up while the mapper is still executing map function calls. The buffer contents are spilled to disk and the buffer is reused. For under-sized mappers, at the end there is an extra merge phase that merges together all existing spills. If combiners are defined then they are applied before spills.

The impact of elasticity on mapper runtime Fig. 1a shows the mapping between normalized mapper runtime (y-axis) and allocated heap size (x-axis) for several Hadoop mappers. We call this mapping the *memory elasticity profile*. The penalties are moderate. For example, an under-sized WordCount mapper is about 1.35x slower than when well-sized. If the same mapper uses a combiner, then the penalty is further reduced (1.15x) because less data is written to disk. The maximum encountered penalty across all mappers is 1.5x.

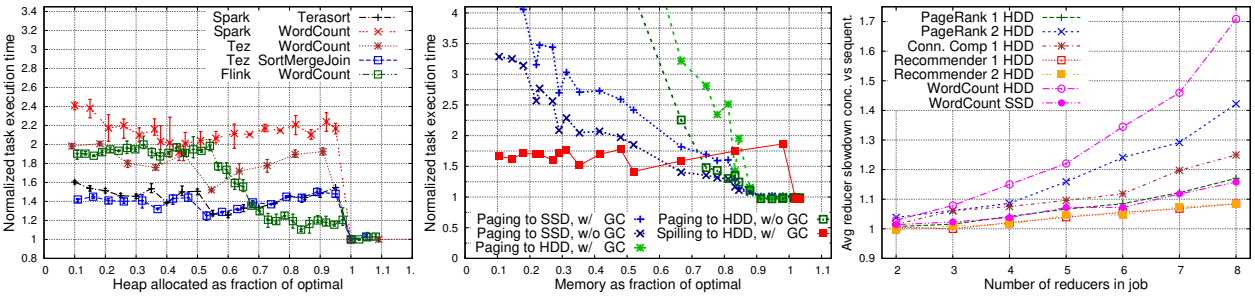
Why penalties are not larger As explained in the introduction, three factors limit the penalties. First, the mapper couples shuffling with CPU-intensive work done by map function calls. Second, disk accesses are efficient as the disk is accessed sequentially. Third, the performance of external merge-sort algorithms remains stable despite significant reductions in shuffle memory.

The shape of the memory elasticity profile The elasticity profile of a mapper resembles a step function. The reason is that under-sized mappers perform an extra merge phase which takes a similar amount of time for many different under-sized allocations.

Modeling memory elasticity for mappers A step function is thus a simple and good approximation for modeling memory elasticity for Hadoop mappers. To build this model two profiling runs are needed, one with an under-sized mapper and one with a well-sized mapper. The runtime of the under-sized mapper can then be used to approximate the mapper runtime for all other under-sized memory allocations.

2.3 Memory elasticity for Hadoop reducers

Elasticity for reducers appears on their input side. Reducers need to read all map outputs before starting the first call to the reduce function. Map outputs are first buffered in memory. For a well-sized reducer this buffer never fills up and it is never spilled. These in-memory map outputs are then merged directly into the reduce functions. For an under-sized reducer the buffer fills up while map outputs are being copied. In this case, the buffer is spilled and reused. The in-memory and on-disk data is then merged and fed to the reduce function of the under-sized reducer.



(a) Elasticity for Spark, Tez and Flink reducers (b) Spilling vs. paging (c) Impact of contention on Hadoop reducers

Figure 2: Memory elasticity for Spark, Tez and Flink jobs (a). Spilling vs. paging (b). Impact of disk contention (c).

The impact of elasticity on reducer runtime Fig. 1b shows the memory elasticity profiles for several Hadoop reducers. The reducer input size ranged from 5GB to 18GB with an average of 10GB. In addition to the jobs in Fig. 1b we also profiled the vector creation part of HiBench's Bayesian Classification. Because this application has many jobs we could not obtain the full elasticity profile for each individual job. Instead, we inferred the maximum penalty for each job using the model described at the end of this subsection. For the 8 distinct jobs we encountered, the maximum penalties are: 1.8x, 1.67x, 2.65x, 1.42x, 3.32x, 1.37x, 1.75x and 1.42x.

Two main insights arise from the results in Fig. 1b. Most importantly, under-sized reducers incur only moderate penalties. Given as little as 10% of ideal memory, 7 of the 10 reducers are between 1.1x and 1.85x slower than ideal. Second, the penalties are comparable for a wide range of under-sized memory allocations. For the WordCount reducer, for 83%, 41% and 10% of ideal memory the penalties are: 1.84x, 1.83x and 1.82x.

Why the penalty varies among reducers We found that the penalty correlates with the complexity of the reduce function. More complex reducers are more CPU-intensive and thus are influenced less by reading inputs from disk. A TeraSort reducer is very simple and shows one of the highest penalties. On the other extreme, the NutchIndexing reducer is complex and shows almost no penalty. To further analyze how reducer complexity influences the penalty we added to the WordCount reduce function a number of floating point operations (FPops) between two randomly generated numbers. Adding 10, 50 and 100 FPops per reduce function call decreased the maximum penalty from 2x to 1.87x, 1.65x and 1.46x.

We also found that the penalty correlates with the number of values corresponding to each key processed by a reducer. A large number of values per key leads to increased penalties because the read-ahead performed by the OS becomes insufficient to bring all keys in memory and thus some reduce calls will read on-disk data. The Mahout recommender uses on average 1500 keys per value for job1 and 15000 keys per value for job2. This explains their larger penalty.

Why penalties are not larger The explanations for reducers are the same as the ones provided for mappers.

Modeling Hadoop reducer behavior An accurate reducer model can be obtained from just two profiling runs: one with under-sized reducers and one with well-sized ones. Given these, the model can infer the penalty for all other under-sized memory allocations. While the profiling runs are application-specific, the model we describe is generally applicable to any Hadoop reducer.

Our model is based on three insights: (a) disk-spilling creates additional penalty on top of the ideal runtime, (b) the penalty is proportional to the amount of data spilled, and (c) the disk rate for reading and writing spills remains constant for a reducer regardless of its memory allocation. We summarize this in the following equation:

$$T(notId) = T + spilledBytes(notId)/diskRate$$

$T(notId)$ is the reducer runtime for an under-sized reducer given $notId$ (not ideal) memory. T is the runtime when that reducer is well-sized. $spilledBytes(notId)$ is the amount of data spilled when being allocated $notId$ memory. Finally, $diskRate$ is the average rate at which the under-sized reducer uses the disk when spilling.

The two profiling runs provide $T(notId)$ and T. Next, $spilledBytes(notId)$ can be computed numerically from the reducer input size, the value of $notId$, and a few Hadoop configuration parameters, thus yielding $diskRate$. Any other $T(notId')$ can be obtained numerically by computing the corresponding $spilledBytes(notId')$ and plugging it in the equation.

Fig. 1c shows the accuracy of our model for the Hadoop reducers profiled in Fig. 1b. The value of $notId$ chosen for the under-sized profiling run was 52% of optimal. Any other under-sized amount would have sufficed. The accuracy of the model is within 5% for most cases.

The shape of the memory elasticity profile We now explain the sawtooth-like shape of the memory elasticity profiles from Fig. 1b. The key insight is that the penalty

is proportional to the amount of spilled data.

The sawtooth peaks are caused by the input-to-buffer ratio of a reducer. When the input is close to a multiple of the buffer, almost all data gets spilled to disk. For example, given a 2.01GB input, two reducers with 500MB and 2GB shuffle buffers, respectively, will each spill 2GB.

There are several cases in which decreasing the memory allocation also decreases the penalty (e.g., WordCount with 52% vs. 83% of ideal memory). This is caused by a decrease in the amount of spilled data. Given a 2GB shuffle buffer and a 2.01GB input size, a reducer spills 2GB to disk but given a 1.5GB shuffle buffer it spills only 1.5GB to disk and keeps 510MB in memory.

One may argue that the static threshold used by Hadoop for spilling is inefficient and that Hadoop should strive to spill as little as possible. In this argument, the reducer with 2.01GB input and a 2GB shuffle buffer would spill 10MB only. Such a proposal actually strengthens the case for memory elasticity as the penalties decrease (due to less spilled data) and can be modeled similarly.

2.4 Elasticity for Spark, Tez and Flink

Fig. 2a shows that memory elasticity also applies to Spark. The reducer input size ranged from 2GB to 22GB with an average of 10GB. For Spark we profiled a task performing a sortByKey operation (TeraSort) and one performing a reduceByKey operation (WordCount). Internally, Spark treats the two cases differently. A buffer is used to store input data for sortByKey and a hashmap for reduceByKey. Both data structures are spilled to disk when a size threshold is reached. Despite the differences both tasks show elasticity.

Fig. 2a shows that the elasticity profile for Spark resembles that of Hadoop reducers. Given the similarities we were able to extend our Hadoop reducer model to Spark sortByKey tasks (TeraSort). The difference between the Hadoop and Spark TeraSort model is that for Spark we also learn an expansion factor from the under-sized profiling run. This is because Spark de-serializes data when adding it to the shuffle buffers. Fig. 1c shows that the accuracy of the Spark model is well within 10%, matching that of Hadoop.

Fig. 2a also shows the memory elasticity profiles for two Tez reducers. The elasticity profile for Tez is similar to those for Spark and Hadoop. We extended our Hadoop reducer model to Tez reducers by accounting for the fact that in Tez, map outputs stored on the same node as the reducer are not added to shuffle memory but are instead read directly from disk. Fig. 1c shows that the accuracy of our Tez model is equally good.

Finally, Fig. 2a also shows one Flink reducer. Flink stands out with its low penalty between 70% and 99% of optimal memory which suggests a different model is needed. We plan to pursue this as part of our future work.

2.5 Spilling vs. paging

Why do frameworks implement spilling mechanisms and do not rely on tried-and-tested OS paging mechanisms for under-sized tasks? To answer, we provisioned Hadoop with enough memory to avoid spilling but configured cgroups such that part of the memory is available by paging to a swapfile. Fig. 2b shows the results for the Hadoop Wordcount reducer. Paging wins when a task gets allocated close to ideal memory (0.7 or more on the x-axis) because it only writes to disk the minimum necessary while Hadoop spills more than necessary. However, spilling beats paging for smaller memory allocations because the task's access pattern does not match the LRU order used by paging. Fig. 2b also shows that paging greatly increases garbage collection (GC) times because the GC touches memory pages in a paging-oblivious manner. We also see that the SSD significantly outperforms the HDD due to more efficient page-sized (4k) reads and writes. Using 2MB transparent huge pages (THP) did not improve results for either the SSD or HDD since THP is meant to alleviate TLB bottlenecks not improve IO throughput.

2.6 Memory elasticity and disk contention

Since memory elasticity leverages secondary storage, it is interesting to understand the impact of disk contention when several under-sized tasks are collocated.

The impact of disk contention depends on how well provisioned the local storage is on nodes relative to computing power. The ratio of cores to disks can give a sense of how many under-sized tasks can compete, in the worst case, for the same disk (a task usually requires at least one core). In current data centers the ratio is low. In [23], the authors mention ratios between 4:3 and 1:3 for a Facebook 2010 cluster. Public provider offerings also have low core to disk ratios. The list of high-end Nutanix hardware platforms [2] shows plenty of offerings with a ratio of less than 2.5:1 and as low as 0.66:1. Nutanix has more than two thousand small and medium size clusters at various enterprises [9].

Nevertheless, not all clusters are equally well provisioned. Thus, we analyzed the degree to which memory elasticity can produce disk contention by varying the number of under-sized Hadoop reducers that spill concurrently to the same disk. We start between 2 and 8 under-sized reducers each within 1 second of the previous. This is akin to analyzing disk contention on nodes with a core to disk ratio ranging from 2:1 to 8:1. We focused on reducers because they spill more data than the mappers (GBs is common).

We measured the slowdown in average reducer runtime when all reducers run concurrently compared to the case where they run sequentially. Fig. 2c shows the results. Several reducers (PageRank job1, Recommender

job1,2) show minimal slowdown (at most 15% degradation for 8 concurrently under-sized reducers). In the other extreme, running 8 under-sized WordCount reducers concurrently leads to a 70% degradation when an HDD is used but that is reduced to just 15% when moving the spills to SSD. In conclusion, disk contention is a manageable potential side effect but should nevertheless be taken into consideration when leveraging elasticity.

2.7 Final considerations

Does elasticity cause increased GC? For Hadoop and Tez reducers, GC times remain constant when tasks are under-sized. For Hadoop mappers, GC times slowly increase as the memory allocation decreases but they remain small in all cases. Overall, Hadoop does a good job of mitigating GC overheads by keeping data serialized as much as possible. For Spark, GC times increase sub-linearly with an increase in task runtime. Interestingly, GC times are a larger portion of task runtime for well-sized tasks because spill episodes limit the amount of data that needs to be analyzed for GC.

Feasibility of modeling Our models for Hadoop, Tez and Spark are based on two profiling runs, one under-sized and one well-sized. Related work shows that a large fraction of jobs in current data centers are recurring, have predictable resource requirements and compute on similar data [14, 20, 5]. Thus, instead of profiling runs, one can use prior runs of recurring jobs. Alternatively, if no prior runs are available, the two profiling runs can be performed efficiently on a sample of the job's input data.

Sensitivity to configuration changes We repeated our experiments on two different hardware platforms (Dual Intel Xeon E5-2630v3 + 40Gb NIC, Dual AMD Opteron 6212 + 10GB NIC), two OSes (RHEL 7, Ubuntu 14.04), three different disk configurations (HDD, SDD, 2*HDD in RAID 0), three IO schedulers (CFS, deadline, noop) and three JVMs (HotSpot 1.7, 1.8, OpenJDK 1.7). The changes did not impact the memory elasticity profiles or the accuracy of our model.

Elasticity of interactive workloads While interactive applications are far less tolerant to increases in runtime, they may still benefit from memory elasticity (e.g., by reducing queuing times in saturated clusters). Furthermore, it can be ensured that latency-sensitive jobs are not impacted negatively by providing additional job constraints (such as execution deadlines).

3 Applying elasticity to cluster scheduling. Case study: Apache YARN

In this section, we explore the benefits that memory elasticity can provide in cluster scheduling by integrating memory elasticity into Apache YARN [25]. We chose YARN because it is very popular and provides a common resource management layer for all popular frameworks tested in §2. Moreover, several recent research efforts from large Internet companies were validated with implementations on top of YARN [11, 21, 17, 18]. In addition, we also discuss how the elasticity principles can be adopted to the Mesos [19] resource manager.

Scheduling using memory elasticity is an NP-hard problem because it contains as a special case NP-hard variants of the RCPSP problem [8], a well-known problem in operations research. Nevertheless, we show that the benefits of memory elasticity can be unveiled even using a simple heuristic.

3.1 Overview

YARN distributes cluster resources to the jobs submitted for execution. A typical job may contain multiple tasks with specific resource requests. In YARN, each task is assigned to a single node, and multiple tasks may run concurrently on each node, depending on resource availability. The scheduler has a global view of resources and orders incoming jobs according to cluster policy (e.g., fair sharing with respect to resource usage).

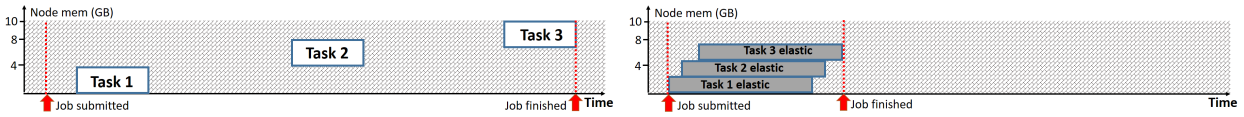
Notation We use the term *regular* to refer to the memory allocation and runtime of well-sized tasks and the term *elastic* for under-sized tasks. We further refer to our elasticity-aware implementation as YARN-ME.

Benefits As previously discussed, memory elasticity trades-off *task execution time* for *task memory allocation*. When applied to cluster scheduling it becomes a trade-off between *task queuing time* and *task completion time*. A task normally has to wait until enough memory becomes available for it but executing it with less memory may reduce or eliminate its waiting time. Since the completion time of a task is the sum of waiting time plus runtime, a significant decrease in waiting time may outweigh an increase in runtime due to elasticity and overall lead to faster task completion times. YARN-ME turns this task-level trade-off into improved job completion time and improved cluster-wide memory utilization.

Fig. 3 illustrates how memory elasticity benefits scheduling using a simple example of a 3-task job scheduled on a single, highly utilized node. Fig. 3a presents a timeline of task execution for vanilla YARN. Tasks 2 and 3 incur queuing times much larger than their execution times. In Fig. 3b, using memory elasticity, the scheduler launches all tasks soon after job submission, resulting in the job completing in less than 30% of its original time, despite its tasks now taking twice as long to execute.

3.2 System design

Two main additions are needed to leverage memory elasticity in YARN.



(a) All tasks *regularly* allocated.

(b) All tasks *elastically* allocated (with less memory).

Figure 3: Example of job completion time reduction for a simple 3-task job and one highly utilized node. In (a) the tasks wait for resources, while in (b) they start almost immediately. The job finishes faster, despite the longer tasks.

Metadata regarding task memory elasticity Reasoning about memory elasticity at the scheduler level leverages additional knowledge about the submitted tasks. The scheduler uses estimates for the regular execution time of a task (ideal duration), the minimal amount of memory for a regular allocation (ideal memory) and the performance penalty for under-sized memory allocations. This metadata is obtained using the profiling techniques from §2.7.

The timeline generator YARN-ME uses a timeline generator to provide an estimate of a job's evolution (the completion times of its tasks and of the whole job). In doing this, it accounts for the expected memory availability in the cluster. The generator simply iterates over all the nodes, adding up the task duration estimates of the executing and queued tasks. Effectively, the generator builds simple timelines for each node, which it then merges to obtain information about each job's timeline. The generator runs periodically, every heartbeat interval, since during such a period all healthy nodes report their status changes. It also runs when a new job arrives or an existing one is prematurely canceled.

3.3 Scheduler decision process

Algorithm 1 presents the decision process of YARN-ME. Lines 8-10 implement the main heuristic underlying YARN-ME (described below). Line 7 also implements disk contention awareness. Additionally, lines 7 and 9 always consider the minimum amount of memory that yields the lowest possible execution time, leveraging the behavior of elasticity described in §2.

Main heuristic YARN-ME aims to reduce job completion time by leveraging memory elasticity. As such, an elastic task cannot be allowed to become a straggler for its respective job. Therefore, *an elastic allocation is made for a task that cannot be scheduled regularly iff its expected completion time does not exceed the current estimated completion time of its job*.

Disk contention awareness As shown in §2.6 scheduling too many elastic tasks concurrently on a node may lead to disk contention. YARN-ME incorporates disk contention awareness. As shown in §2.6, obtaining the task metadata involves computing the disk bandwidth required by an elastic task. YARN-ME allocates a portion of each node's disk bandwidth for elastic tasks.

Algorithm 1 YARN-ME decision process pseudocode.

```
1:  while JOB QUEUE is not empty do
2:      for all N in NODES do
3:          J ← N's reserved job or next job in JOB QUEUE
4:          T ← next task of J
5:          if T regularly fits on N then
6:              allocate T on N, regular
7:          else if T elastically fits on N then
8:              get TIMELINE GENERATOR info for J
9:              if T elastically finishes before J then
10:                 allocate T on N, elastic
11:         if T was allocated then
12:             unreserve N if reserved
13:             resort the JOB QUEUE
14:         else
15:             reserve N for J, if not already reserved
```

It conservatively prohibits the scheduling of new elastic tasks on nodes where this portion would be exceeded.

Node reservations In YARN, if a node has insufficient resources to satisfy the job at the head of the queue, no allocation is performed, and the node is reserved for that job. While this job still has pending tasks, no other jobs can schedule tasks on the reserved node. This helps mitigate resource starvation by ensuring that the tasks of jobs with large memory requirements also get the chance to be scheduled. To account for this, we adjusted the *timeline generator* to take reservations into account when building its estimates. Additionally, YARN-ME allows tasks of other jobs to be allocated on a reserved node, but only if this does not hinder the tasks of the reserved job.

Additional constraints Schedulers may set additional constraints for their jobs, such as running on a data-local node only, or forcing certain tasks to start only after others have completed. Our design is orthogonal to this and only requires tweaking of the *timeline generator*.

4 Discussion: Mesos

Other schedulers beyond YARN can also be extended to use memory elasticity. We next review the main differences between Mesos [19] and YARN and argue that they do not preclude leveraging memory elasticity in Mesos.

Queuing policy Mesos uses Dominant Resource Fairness (DRF) [15], a multi-resource policy, to ensure fairness. Thus, the job queue may be sorted differently compared to YARN's policies. This does not restrict

memory elasticity as it only dictates job serving order.

Decision process Mesos decouples scheduling decisions from node heartbeats. Thus, a job may be offered resources from several nodes at the same time. This does not restrict memory elasticity since the job needs to consider each node from the offer separately (a task can only run on one node), so memory elasticity can be applied for every node in the offer.

Global vs. local decisions Mesos gives jobs the ability to accept or reject resource offers while YARN decides itself what each job receives. Thus, in Mesos, jobs can decide individually whether to use elasticity or not. If a decision based on global cluster information (like in YARN) is desired, jobs can express constraints (locality, machine configuration) with Mesos filters that can be evaluated by Mesos before making resource offers.

5 Cluster experiments

We next showcase the benefits of memory elasticity by comparing YARN-ME to YARN.

5.1 Methodology

Setup We use a 51-node cluster (50 workers and 1 master), limiting the scheduler to 14 cores per node (out of 16 cores we reserve 2 for the YARN NodeManager and for the OS) and 10GB of RAM. The exact amount of RAM chosen is not important (we could have chosen any other value). What is important is the ratio of ideal task memory requirements to node memory. Each node has one 2 TB SATA HDD. YARN-ME was implemented on top of Apache YARN 2.6.3 [25]. Disk spills use Direct I/O so that the OS buffer cache does not mask performance penalties due to elasticity.

We ran WordCount, PageRank and Mahout Item Recommender Hadoop applications. We chose them because they represent small, medium and large penalties encountered for reducers in §2 (mapper penalties span a much smaller range than reducers and are lower). We configured the jobs as described in Table 1. We executed each type of application separately (homogeneous workload) and all applications simultaneously (heterogeneous workload). For the homogeneous workloads, we varied the number of concurrent runs for each type of application. The start of each run is offset by the inter-arrival (IA) time indicated in Table 1. The IA time is chosen proportionally to application duration such that map and reduce phases from different jobs can overlap.

For each application we first perform one profiling run using ideal memory to obtain the ideal task runtime. We multiply this by the penalties measured in §2 to obtain task runtimes for various under-sized allocations.

Metrics We compare average job runtime, trace makespan and average duration of *map* and *reduce* phases. By *job runtime* we mean the time between job submission and the end of the last task in the job. Similarly, a map or reduce phase represents the time elapsed between the first request to launch such a task and the finish time of the last task in the phase. Each experiment is run for 3 iterations, and we report the average, minimum and maximum values.

5.2 Experiments

Benefits for memory utilization Fig. 4a shows the benefits of using memory elasticity on both cluster utilization and makespan, for an execution of 5 Pagerank runs. YARN-ME successfully makes use of idle memory, bringing total memory utilization from 77% to 95%, on average, and achieving a 39% reduction in makespan. By comparing the results in Figs. 4a and 4c we also see that the gain in job runtime is proportionally much higher than the amount of memory reclaimed. Fig. 4b shows how YARN-ME assigns the memory slack to tasks.

Benefits for homogeneous workloads We next show that YARN-ME can provide benefits for the jobs in Table 1. Figs. 4c, 5a and 5b show the improvement of YARN-ME compared to YARN vs. the number of runs. YARN-ME's benefits hold for all jobs.

We find that the Recommender, which has the highest penalties we have observed for reducers, achieves up to 48% improvement. We also find that mappers always benefit noticeably from elasticity, a direct consequence of their modest penalties. Pagerank's lower-penalty reducers yield an improvement of 30% even for a single concurrent run, peaking at 39% with 5 runs. Wordcount achieves a peak improvement of 41%, despite reducer gains being lower due to higher penalties. The reduction in average job runtime steadily increases across runs. For 3 concurrent Wordcount runs, the number of reducers leads only one out of the 3 jobs to be improved, but the map phase still reaches improvements of 46%.

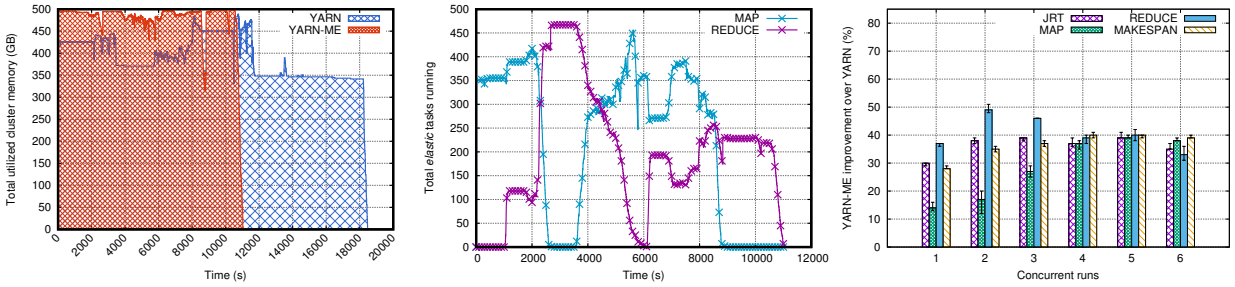
Benefits for heterogeneous workloads YARN-ME achieves considerable gains even under a heterogeneous workload composed of all the jobs from Table 1. We start 5 jobs at the same time (1 Pagerank, 1 Recommender and 3 Wordcount) and then submit a new job every 5 minutes, until we reach a total of 14 jobs (3 Pagerank, 3 Recommender and 8 Wordcount). Each job is configured according to Table 1. Fig. 5c shows overall improvement and breakdown by job type. YARN-ME improves average job runtime by 60% compared to YARN. The *map* phase duration is reduced by 67% on average overall, and by up to 72% for all Recommender jobs.

6 Simulation experiments

We use simulations to evaluate YARN-ME's benefits and its robustness to mis-estimations on a wide range of workload configurations.

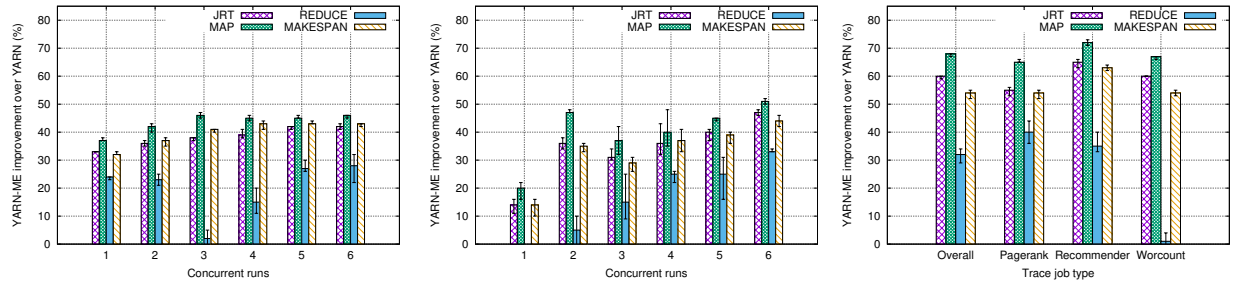
Application	Jobs	Input GB	maps	reduces	Penalties				Memory GB				Inter-arrival (IA) time
					1st job		2nd job		1st job		2nd job		
					map	reduce	map	reduce	map	reduce	map	reduce	
Pagerank	2	550	1381 / 1925	275	1.3	1.22	1.25	1.75	1.7	3.7	1.5	6.8	120s
WordCount	1	540	2130	75	1.35	1.9	-	-	1.7	5.4	-	-	30s
Recommender	2	250	505 / 505	100	1.3	2.6	1.3	3.3	2.4	2.8	2.4	3.8	120s

Table 1: Characteristics of the evaluated applications.



(a) Cluster memory utilization – 5 runs (b) *elastic* tasks – 5 runs (c) Improvement over YARN

Figure 4: YARN-ME vs. YARN for running Pagerank on 50 nodes. Fig. 4a shows the timeline of cluster memory utilization. Fig. 4b shows the timeline of tasks scheduled *elastically*. Fig. 4c reports improvement w.r.t. *average job runtime* (JRT), *average job phase time* (map, reduce), and *makespan*.



(a) WordCount (b) Recommender (c) Mixed job trace – 14 jobs

Figure 5: Improvement of YARN-ME over YARN for various applications on 50 nodes w.r.t. *average job runtime* (JRT), *average job phase runtime* (map, reduce), and *makespan*. We report average, min. and max. over 3 iterations. Fig. 5c reports results for a mixed trace of jobs: 3x Pagerank, 3x Recommender, 8x Wordcount.

6.1 Simulation Methodology

Simulator We built DSS (Discrete Scheduler Simulator), a discrete-time simulator for YARN, and we made the code publicly available [1]. In DSS, simulated tasks do not perform computation or I/O. The tasks are simulated using task start and task finish events. We simulate a cluster with 16 cores and 10GB of RAM per node. Memory is assigned to tasks with a granularity of 100MB. Jobs are ordered according to YARN's fair-scheduling policy [1]. Each task uses 1 core. The minimum amount of memory allocatable to a task is set to 10% of its ideal requirement. We use a 100-node cluster to perform a parameter sweep but also show results for up to 3000 nodes.

Simulation traces A trace contains for each job: the job submission time, the number of tasks, the ideal amount of memory for a task, and task duration. Each job has one parallel phase. Job arrivals are uniformly random between 0 and 1000s. The other parameters are varied according to either a uniform or an exponential random distribution. We use 100-job traces but also show results for up to 3000 jobs.

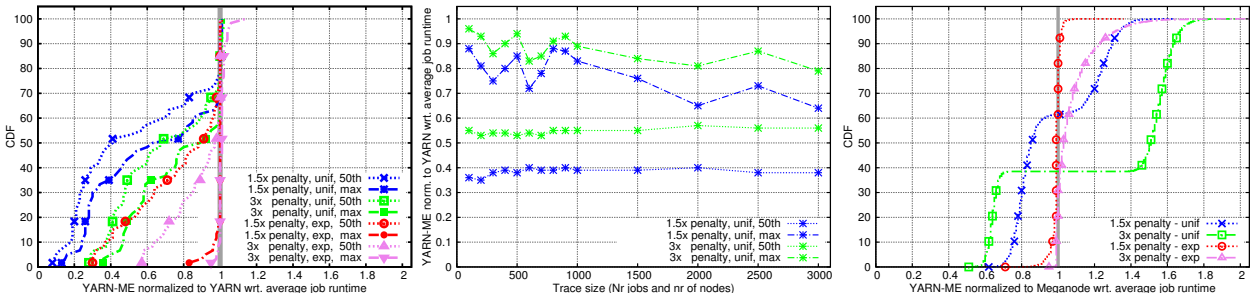
Modeling elasticity Since the simulated jobs have only a single phase we only use the reducer penalty model from §2. We show results for 1.5x and 3x penalties, to cover the range of penalties measured in §2.

Metrics We use average job runtime to compare the different approaches.

6.2 Simulation experiments

YARN-ME vs. YARN We perform a parameter sweep on 3 trace parameters: memory per task, tasks per job and task duration. Table 2 shows the different parameter ranges. We keep the min constant and vary the max within an interval to perform the sweep. This gives us a

[1] https://github.com/epfl-labos/DSS



(a) YARN-ME vs. YARN (avg job runtime) (b) Impact of scaling trace size (c) YARN-ME vs. Meganode (avg job runtime)

Figure 6: CDFs of average job runtime in YARN-ME vs. YARN (a) and vs. the idealized Meganode (c). YARN's benefits hold for traces of varying size (b).

range for each parameter, which is then varied independently of the others. We draw the values from a uniform or exponential random distribution. We perform 100 runs for each combination of 3 ranges (one for each parameter) and show the median and the maximum (worst-case) results for normalizing YARN-ME to YARN in Fig. 6a. We use 100-job traces on 100 nodes.

dist	tasks / job		task mem (GB)		task duration (s)	
	min	max	min	max	min	max
unif	1	[200,400]	1	[2,10]	1	[200,500]
exp	1	[20,220]	1	[2,10]	50	[100,500]

Table 2: Trace parameter sweep ranges.

The uniform distribution yields bigger benefits because it leads to more memory fragmentation in YARN. As expected, YARN-ME's improvements are larger if penalties are lower. The cases in which YARN-ME does not improve on YARN are either when the cluster utilization is very low or when most tasks have very small memory requirements. In such cases, memory elasticity is less beneficial. Nevertheless, for 3x penalty and a uniform distribution, 40% of the configurations have a ratio of YARN-ME to YARN of at most 0.7.

Fig. 6b shows the behavior of one uniform trace in a weak scaling experiment. We scale the trace and cluster size simultaneously from 100 to 3000. The benefits of YARN-ME hold despite the scaling.

The need for elasticity (YARN-ME vs. Meganode) We next show that YARN-ME yields improvements beyond the reach of current elasticity-agnostic schedulers. We compare against an idealized scenario (called Meganode) which serves as an upper-bound for elasticity-agnostic solutions that improve average job runtime. The Meganode pools all cluster resources into one large node with a memory and core capacity equal to the aggregate cluster-wide core and memory capacity available for YARN-ME. Thus, the Meganode does away with machine-level memory fragmentation. Meganode uses a shortest remaining job first (SRJF) scheduling policy because that is known to improve average job runtime. However, in using SRJF, YARN-ME obeys the existing fairness policy whereas Meganode does not.

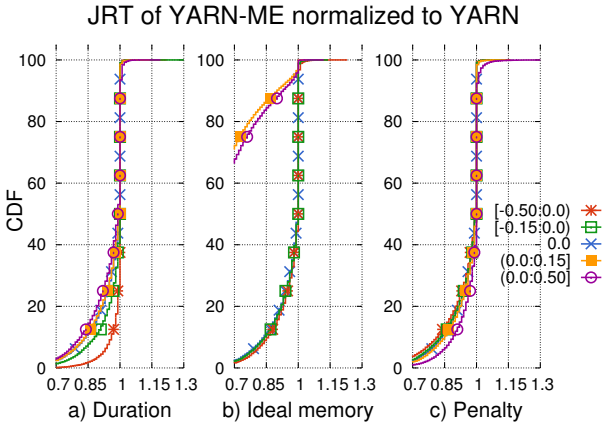


Figure 7: Sensitivity to mis-estimations. 3x penalty.

Fig. 6c compares the averge job runtime for Meganode and YARN-ME on 20.000, 100-job traces on 100 nodes. While it is expected that Meganode wins in many cases, YARN-ME beats Meganode for 40%-60% of the cases for the uniform trace and for 20% of the exponential trace for 1.5x penalty. YARN-ME gains because it turns even small amounts of memory fragmentation into an opportunity by scheduling elastic tasks.

Sensitivity to mis-estimations Further, we show that YARN-ME is robust to mis-estimations. We generate 20,000 traces with each of the trace parameters (memory per task, tasks per job, and task duration) following an exponential random distribution, within bounds of $[0.1,10]$ GBs, $[1,100]$ tasks, and $[50,500]$ seconds. We simulate mis-estimations by altering the duration, ideal memory, and performance penalty of tasks for both regular and elastic allocations. This forces the scheduler to make decisions based on imperfect information. We change each parameter fractionally by a uniformly random factor in the intervals of $(0,0.15]$, and $(0,0.5]$ (0.15 represents a 15% higher value). The former interval represents the worst-case deviation of our model in Fig. 1c,

while the latter is an extreme example chosen to stress YARN-ME. We present both positive and negative mis-estimations. Fig. 7 presents the ratio between average job completion time with YARN-ME and YARN, for a 3x penalty – one of the highest penalties measured in §2.

Sensitivity to task duration mis-estimation YARN-ME is robust to task duration mis-estimation, which can occur due to system induced stragglers or data locality. The timeline generator of the simulator bases its information on task durations from the trace. We alter each actual task runtime by a different factor.

For $[-0.15, 0.5]$, YARN-ME achieves gains similar to the scenario without mis-estimations on all traces. Even for the very large $[-0.5, 0)$ mis-estimations, the gains are still comparable, with only $\sim$35% of the traces reporting at most 10% lower gains. This is due to tasks being shorter than the timeline generator expects, resulting in a few elastic tasks exceeding the estimated job finish time.

Sensitivity to model mis-estimations YARN-ME is also robust to model mis-estimations, which may occur during profiling. We change task memory (Fig. 7b) and penalty (Fig. 7c) by a different value for each job.

YARN-ME improves by up to 45% in the case of positive mis-estimation of ideal memory (Fig. 7b). In this case, all tasks (in both YARN and YARN-ME) spill data to disk and become penalized tasks. However, penalties in YARN-ME are lower because YARN-ME can choose the under-sized allocation that minimizes penalty while YARN lacks this capability. Negative mis-estimation of ideal memory has negligible impact.

In the case of penalty mis-estimation (Fig. 7c), only the $(0, 0.5]$ runs exhibit gains reduced by at most 4%. This is due to the scheduler being more conservative since it perceives elastic tasks as taking longer.

7 Related Work

Current schedulers do not leverage memory elasticity. Next, we review the most related mechanisms employed by current schedulers.

Tetris [16] improves resource utilization (including memory) by better packing tasks on nodes. It adapts heuristics for the multi-dimensional bin packing problem to the context of cluster scheduling. However, it only schedules a task on a node that has enough memory available to cover its estimated peak memory requirements.

Heracles [22] aggressively but safely collocates best-effort tasks alongside a latency critical service. It does this by dynamically managing multiple hardware and software mechanisms including memory. However, Heracles only considers RAM bandwidth and not capacity.

Apollo [7] is a distributed scheduler that provides an opportunistic scheduling mode in which low priority tasks can be scheduled using left-over memory unused by normal priority tasks. Normal priority tasks are scheduled only if their resource demands are strictly met. Apollo has no principled way of reasoning about the performance implications of opportunistic allocations nor does it provide a decision mechanism about when such allocations are useful. Borg [26] provides similar capabilities with a centralized design.

Quasar [12] leverages machine-learning classification techniques to reason about application performance with respect to scale-up allocations. A greedy algorithm places tasks starting with nodes that give the best performance satisfying application SLOs and improving resource utilization. Quasar does not identify nor discusses memory elasticity.

ITask [13] is a new type of data-parallel task that can be interrupted upon memory pressure and have its memory reclaimed. The task can then be resumed when the pressure goes away. ITask is a system-level mechanism that uses preemption to mitigate unmanageable memory pressure before it can hurt system performance. Memory elasticity can work in tandem with ITask, since elastic tasks will need less time to spill, and thus can be preempted and resumed faster than regular tasks.

8 Conclusion

The main contribution of this paper is identifying, quantifying and demonstrating memory elasticity, an intrinsic property of data-parallel workloads. Memory elasticity allows tasks to run with significantly less memory than ideal while incurring only a moderate performance penalty. We show that memory elasticity is prevalent in the Hadoop, Spark, Tez and Flink frameworks. We also show its predictable nature by building simple models for Hadoop and extending them to Tez and Spark. Applied to cluster scheduling, memory elasticity helps reduce task completion time by decreasing task waiting time for memory. We show that this can be transformed into improvements in job completion time and cluster-wide memory utilization. We integrated memory elasticity into Apache YARN and showed up to 60% improvement in average job completion time on a 50-node cluster running Hadoop workloads.

Acknowledgements We thank our anonymous reviewers and our shepherd, Christina Delimitrou, for valuable feedback and advice. We thank Laurent Bindschaedler, Diego Didona, Christos Gkantsidis, Ashvin Goel, Sergey Legtchenko, Baptiste Lepers, T. S. Eugene Ng, Amitabha Roy, and Yiting Xia for feedback and suggestions on earlier drafts of the paper and insightful discussions. We also thank Rolf Möhring, Roel Leus, and Sigrid Knust for valuable explanations regarding RCPSP, and Adrian Popescu for providing the TPC-DS queries.

References

[1] Hadoop MapReduce Next Generation - Fair Scheduler. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[2] Hardware Platform Specifications. http://www.nutanix.com/products/hardware-platforms/.

[3] Improving Sort Performance in Apache Spark: It's a Double. http://blog.cloudera.com/blog/2015/01/improving-sort-performance-in-apache-spark-its-a-double/.

[4] The Bigdata Micro Benchmark Suite. https://github.com/intel-hadoop/HiBench.

[5] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing data-parallel computing. In *Proc. NSDI 2012*.

[6] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters. In *Proc. USENIX ATC 2014*.

[7] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *Proc. OSDI 2014*.

[8] P. Brucker, A. Drexi, R. Moerhing, K. Neumann, and E. Pesch. Resource-constrained project scheduling: Notation, classification, models, and methods. In *European Journal of Operational Research, Volume 112, Issue 1, 1 January 1999, Pages 3–41*.

[9] I. Cano, S. Aiyar, and A. Krishnamurthy. Characterizing private clouds: A large-scale empirical analysis of enterprise clusters. In *Proc. SoCC 2016*.

[10] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. In *Proc. VLDB 2012*.

[11] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based scheduling: If you're late don't blame us! In *Proc. SoCC 2014*.

[12] C. Delimitrou and C. Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proc. ASPLOS 2014*.

[13] L. Fang, K. Nguyen, G. Xu, B. Demsky, and S. Lu. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *Proc. SOSP 2015*.

[14] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proc. EuroSys 2012*.

[15] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proc. NSDI 2011*.

[16] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *Proc. SIGCOMM 2014*.

[17] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *Proc. OSDI 2016*.

[18] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *Proc. OSDI 2016*.

[19] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proc. NSDI 2011*.

[20] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *Proc. SIGCOMM 2015*.

[21] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards automated SLOs for enterprise clusters. In *Proc. OSDI 2016*.

[22] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proc. ISCA 2015*.

[23] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics framework. In *Proc. NSDI 2015*.

[24] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proc. SoCC 2012*.

[25] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves,

J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proc. SOCC 2013*.

[26] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proc. EuroSys 2015*.

[27] 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 *Proc. NSDI 2012*.

[28] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J. Y. Li, W. Lin, J. Zhou, and L. Zhou. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In *Proc. NSDI 2012*.

Popularity Prediction of Facebook Videos for Higher Quality Streaming

Linpeng Tang*, Qi Huang♭, Amit Puntambekar♭, Ymir Vigfusson†, Wyatt Lloyd‡,♭, Kai Li*

*Princeton University, †Emory University/Reykjavik University, ‡University of Southern California, ♭Facebook Inc.

Abstract

Streaming video algorithms dynamically select between different versions of a video to deliver the highest quality version that can be viewed without buffering over the client's connection. To improve the quality for viewers, the backing video service can generate more and/or better versions, but at a significant computational overhead. Processing all videos uploaded to Facebook in the most intensive way would require a prohibitively large cluster. Facebook's video popularity distribution is highly skewed, however, with analysis on sampled videos showing 1% of them accounting for 83% of the total watch time by users. Thus, if we can predict the future popularity of videos, we can focus the intensive processing on those videos that improve the quality of the most watch time.

To address this challenge, we designed Chess, the first popularity prediction algorithm that is both scalable and accurate. Chess is scalable because, unlike the state-of-the-art approaches, it requires only constant space per video, enabling it to handle Facebook's video workload. Chess is accurate because it delivers superior predictions using a combination of historical access patterns with social signals in a unified online learning framework. We have built a video prediction service, ChessVPS, using our new algorithm that can handle Facebook's workload with only four machines. We find that re-encoding popular videos predicted by ChessVPS enables a higher percentage of total user watch time to benefit from intensive encoding, with less overhead than a recent production heuristic, e.g., 80% of watch time with one-third as much overhead.

1 Introduction

Video is increasingly a central part of people's online experience. On Facebook alone, there are more than 8 billion video views each day [2]. Clients stream these videos by progressively downloading video chunks from a provider according to an adaptive bitrate (ABR) [33, 39] algorithm. ABR algorithms strive to dynamically select the version of a video with the highest bitrate a connection can sustain without pausing. Higher bitrates provide higher quality, but are larger and thus require clients to have higher-bandwidth connections. The different versions of the video used by ABR algorithms are typically generated when a video is uploaded [3]. Generating the different versions for the large volumes of videos uploaded to Facebook each day requires a large fleet of servers.

There is a trade-off between the amount of computation spent processing a video to prepare it for streaming and the quality of experience for viewing that video. Videos uploaded to Facebook are by default encoded to a small number of standard versions with FFmpeg [16]. However, investing in more computation can improve playback experience by improving or increasing the choices for the ABR algorithm. First, more computation can improve the choices by further compressing a video at a fixed quality. For instance, Facebook's QuickFire engine [1] uses up to 20× the computation of the standard encoding to produce a version of the video with similar (or higher) quality that is ~20% smaller than the standard encoding. Second, more computation can increase the choices for the streaming algorithm by generating more versions of the video at different bitrates. In both cases, added computation increases the highest quality version of a video that can be streamed for some users.

Unfortunately, it is infeasible to compute the highest-quality encodings for all videos. Using QuickFire and increasing the number of versions of each video, for example, would require a fleet at several tens the scale of the already large processing fleet at Facebook. Fortunately, video popularity is highly skewed, with 1% of the videos accounting for over 80% of the *watch time*, i.e., the time users spend viewing video. This skew enables us to achieve most of the quality improvement with only a fraction of the computation by generating the highest-quality encodings for only the most popular videos.

The challenge in exploiting this insight is in scalably and accurately predicting the videos that will become popular. State of the art popularity prediction algorithms [9, 10, 45] are accurate but do not scale to handle the Facebook video workload because they keep per-video state that is linear in its past requests. Simple heuristics that exploit information from the social network scale, but are not accurate. For example, predicting popular videos based on owner like count requires 8× more resources to cover 80% of watch time than what would be needed with clairvoyant predictions, which only runs QuickFire encoding on videos with the largest future watch time.

We overcome this challenge with Chess—Constant History, Exponential kernels, and Social Signals—the

first scalable and accurate popularity prediction algorithm. Chess is scalable because it uses constant per-video state, needing only ~20GB to handle the Facebook video workload. Chess is accurate: it outperforms even the non-scalable state-of-the-art algorithm. Two key insights led to Chess. First, we approximate the history of all de-identified past accesses to a video with exponentially-decayed watch time (§4.1) in a few fixed-size time windows, each of which is not highly accurate but small and fast to compute. Second, we combine those constant sized historical features through a continuously updated neural network model to obtain state-of-the-art accuracy, and then further improve it by leveraging social network features—e.g., the like count of video owner—while remaining scalable.

We validate Chess's scalability by building ChessVPS, a video prediction service based on Chess, that requires only four machines to provide popularity prediction for all of Facebook's videos. ChessVPS has been deployed, providing query-based access to new predictions updated every ten minutes, although its predictions are not yet used to inform encoding choices in production.

Our evaluation compares Chess against the state-of-the-art non-scalable prediction algorithms, simple scalable heuristics, and a clairvoyant predictor using traces of Facebook's video workload. We find Chess delivers higher accuracy than all achievable baselines, and provides QuickFire-encoded videos for more user watch time with less re-encoding. Compared to the heuristic currently used in production, Chess improves the watch time coverage of QuickFire by 8%–30% using same CPU resources for re-encoding. To cover 80% of watch time, Chess reduces the overhead from 54% to 17%.

The contributions of this paper include:

- The case for video popularity prediction services to improve streaming quality. (§3)
- The design of Chess, the first scalable and accurate popularity prediction algorithm. (§4)
- The implementation of ChessVPS, a prediction service for Facebook videos that uses only four machines. (§5)
- An evaluation using Facebook's workload that shows Chess achieves state-of-the-art prediction accuracy, and delivers high watch time coverage for QuickFire with low CPU overhead from re-encoding. (§6)

2 Background

The workflow of videos on Facebook, which starts with an upload and finishes with streaming, is shown in Figure 1. When a video is uploaded, it is immediately encoded with the H.264 codec to a few standard versions for streaming [40]. The encoded files are durably stored in a backend [5, 32]. In addition, the original upload is kept

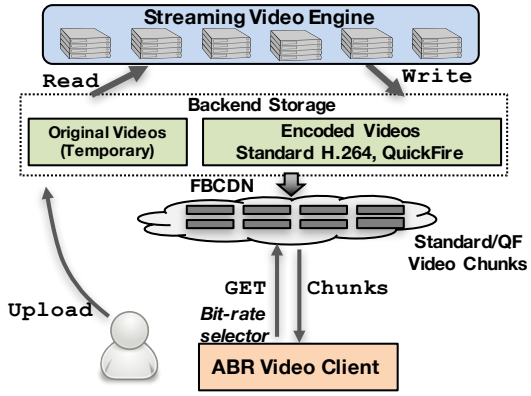


Figure 1: The workflow of videos on Facebook. Additional processing in the Streaming Video Engine can lead to higher quality video delivery to the client by giving the ABR algorithm better choices.

for several days during which it can be re-encoded with QuickFire, used to generate more versions, or both.

Videos are shown to users by a player that downloads progressive chunks of the video from a content distribution network [22, 38]. The player dynamically tries to stream the highest quality version of a video it can without pausing using an ABR algorithm [33, 39]. There are a variety of ABR algorithms [23, 24, 25, 42], but they typically estimate the bandwidth of a user's connection and then select the largest bitrate that is less than that bandwidth.

Generating additional bitrate versions of a video thus improves quality for some users. For example, consider two versions of a video with bitrates of 250 Kbps and 1 Mbps. Generating a third version with a bitrate of 500 Kbps would improve quality for all users with bandwidth between 500 Kbps and 1 Mbps. This is one way additional processing can yield higher-quality video streaming.

Another way to improve video quality is by generating more compressed versions of a video that yield similar or higher video quality at a lower bitrate. FFmpeg's H.264 encoding offers several preset parameters that range from "ultrafast" to "veryslow". Moving to slower encodings yields more compressed versions with the same quality. Facebook's QuickFire [1, 41] technology provides a more extreme trade-off. It intelligently tries many encodings for each chunk of a video, and picks the smallest one with similar or higher quality—client-side decoding is unaffected because each chunk is H.264 compatible. QuickFire can be configured to try 7–20 encodings; we use 20 in this work due to its higher compression.

We quantified this processing/bitrate trade-off for the FFmpeg presets and QuickFire for 1,000 randomly selected videos uploaded to Facebook in one month of 2016. The results of this experiment confirmed that more processing can be used to find better-compressed versions of a video at the same quality. In particular, using QuickFire

takes 20× the processing of "veryslow"[1] but yields a 21% reduction in bitrate for the same quality. This in turn increases the quality of video for some users. For example, consider a 1 Mbps "veryslow" encoding. Generating the QuickFire encoding would yield the same quality at ~800 Kbps. Users with bandwidth between 800 Kbps and 1 Mbps could then stream this higher quality version.

More processing improves the quality of videos that users can stream. Maximally processing all videos would require increasing the already huge number of processing machines by 1-2 orders of magnitude, which is infeasible. Our goal in this paper is to instead extract most of the benefit of using the maximum processing on all videos, but without requiring a substantially larger fleet of machines. We next explain how a scalable and accurate video popularity prediction service helps meet this objective.

3 Motivation and Challenges

This section makes the case for a video popularity prediction service and lays out the challenges of building one, including the need to be quick, accurate, and scalable.

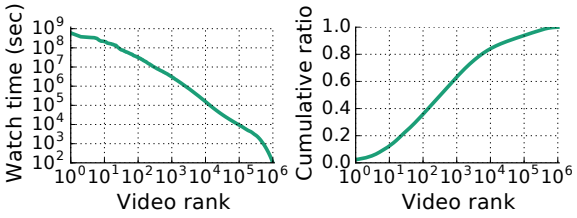
3.1 High Skew Motivates Prediction

Predicting the popularity of videos is compelling because it can guide more processing to where it can do the most good. A small core of videos in Facebook's workload account for most of the time spent watching videos. Thus, if we know what videos will be watched the most in the future, we can focus additional processing on them.

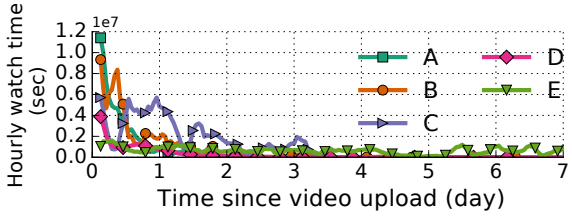
Figure 2a quantifies the skew of Facebook's video workload with the watch time of 1 million randomly sampled videos in one month. The left sub-figure shows the watch time of each unique video, ordered by popularity rank in a log-log scale. For example, the most popular video in the sampled trace has 13 years of watch time in one month, while the $10,000^{\text{th}}$ most popular video out of the million is watched for 42 hours. The shown distribution of watch times follows a power-law with exponent $\alpha = 1.72$. (Related work has shown that access to Facebook photos also follows a power-law distribution with $\alpha = 1.84$ [22].)

The right sub-figure of 2a shows the potential benefit from exploiting this skew. The cumulative ratio of video watch time represented by videos with a given rank or higher is depicted. For example, the top 0.1%/1% of videos account for 62%/83% of the watch time, respectively. Thus, if we use the maximum processing on only 1% of videos we would benefit from increased streaming quality for over 80% of all video watch time. The cumulative watch time ratio is an upper bound on the benefits of popularity prediction because it ranks videos based on their exact accesses, i.e., it represents the benefit from having perfect predictions at the time a video is uploaded.

[1]We could not directly measure the processing time of QuickFire so we approximate it as 20× that of "veryslow" because it encodes each chunk of the video ~20 times



(a) Watch time distribution of sampled videos



(b) Hourly watch time of five example videos

Figure 2: Facebook videos access patterns.

3.2 Prediction is Challenging

The difficulty in exploiting the skew lies in being able to quickly, accurately, and scalably predict the popularity of individual videos. Prediction needs to be quick so not many views of the video are missed while waiting for prediction results. Prediction needs to be accurate so computation is spent on videos that reap the most benefit. Finally, prediction needs to be scalable so it can handle video workloads at a global scale like Facebook.

To motivate each of these points, we manually examined the access pattern of 25 videos in the one month trace with rank 10,000–10,024, i.e., they are near the cut-off for the top 1% of popularity and all have a similar total watch time. Figure 2b shows the access patterns of 5 representative videos. The other 20 videos have access patterns that resemble one of the depicted patterns.

The Need for Quick Prediction The video access pattern peaks quickly for videos A – D in Figure 2b. This indicates we need our video prediction service to run quickly. If our prediction takes longer than the interval between when a video is uploaded and when it peaks, then much of the watch time will have already taken place when the prediction is ready. To further demonstrate this point, we analyzed the full one month trace and found that the most popular 1/4/16 hours of each video accounts for 6.3%/19%/29% of watch time. Previous work on video popularity [18, 36] considered popularity on a daily basis. Such methods, if applied on our workload, would have a large delay in prediction and would miss a significant portion of the total watch time. Instead, we aim for quick predictions on the order of minutes.

The Need for Accurate Prediction The variety of access patterns in Figure 2b suggests that accurately predicting future watch time will be challenging. Prediction needs to be accurate so additional computation is used where it will be the most useful. Using simple heuristics based on features from the social network is quick, but unfortunately is not accurate. For instance, a recent production heuristic was to re-encode a video if the like count of the owner exceeded 10,000. As our evaluation in Section 6 shows, this heuristic is inaccurate: it requires re-encoding 8× as many videos as a clairvoyant predictor to cover 80% of the video watch time. Our goal is to provide predictions with higher accuracy so higher watch time coverage can be achieved with fewer resources.

The Need for Scalable Prediction Video popularity prediction for Facebook must be scalable because there are tens of millions of videos uploaded each day. Identifying popular videos thus requires predicting the popularity of a large active set of videos. In the video prediction service described in Section 5 we track 80 million videos. The previous state of the art in popularity prediction, SEISMIC, is accurate but unfortunately does not scale to our workload because it stores the timestamp and watch time of each past request. This linear per-video state would require ~10TB of memory to make predictions for 80 million videos, and methods requiring more features per request [10] have an even larger memory usage.

4 The Chess Prediction Algorithm

Achieving high watch time coverage through additional processing requires quick, accurate, and scalable prediction of video popularity. This section describes the core of Chess, the novel prediction algorithm we designed with these goals in mind. We focus on three key features:

1. Harnessing past access patterns with constant space and time overhead.
2. Combining different features in a unified model.
3. Efficient online training using the recent access data.

4.1 Utilizing Past Access Patterns with EDWT

A common theme in popularity prediction is exploiting past access patterns [13, 36, 43, 45]. The state of the art approaches do so by modeling behavior as a self-exciting process that predicts future accesses based on all past accesses. A past access at time t is assumed to provide some *influence* on future popularity at time τ, as modeled by a kernel function $\phi(\tau - t)$. The kernel function, ϕ, is a probability density function defined on $[0, +\infty)$, and it is commonly chosen to be a decreasing function, so that a session's influence is initially high and gradually converges to zero over time.

Self-exciting processes predict future popularity—i.e., watch time—based on the sum of the influence of all past requests from the current time to infinity. Let i be an index over the past viewing sessions of a video. Let t_i and x_i be the corresponding timestamp and watch time, respectively of the session. Then, for the purposes of ranking different videos, the total future watch time for i is modeled as

$$\tilde{F}(t) = \sum_{t_i \le t} \int_{\tau}^{+\infty} x_i \phi(\tau - t_i) d\tau.$$

One key insight in Chess is using a kernel that allows for efficient updates to popularity predictions. Previous popularity prediction algorithms used power-law kernels that provide high accuracy predictions, but require each new prediction to compute over all past accesses [13, 45]. This requires storage and computation linear in the past requests to each video, which is not feasible in our setting. In contrast, we set ϕ to be the exponential kernel, or $\phi(t) = \frac{1}{w} \exp(-t/w)$, where w represents a time window modeling how long past requests' influence lasts into the future. Such a kernel allows us to simplify the computation of a new prediction to only require the last prediction, $\tilde{F}$, and its timestamp, u, which drastically reduces the space and time overhead. Below is the simplified update rule for a new session with watch time x beginning at time t with a previous session having occurred at time $u < t$. The resulting prediction is the *exponentially decayed watch time* (EDWT):

$$\begin{aligned}
\tilde{F}(t) &= \sum_{t_i \le t} x_i \int_{t}^{\infty} \phi(\tau - t_i) d\tau \\
&= \frac{x}{w} + \sum_{t_i \le u} x_i \exp\left(\frac{-(t - t_i)}{w}\right) \\
&= \frac{x}{w} + \exp\left(\frac{-(t-u)}{w}\right) \sum_{t_i \le u} x_i \exp\left(\frac{-(u - t_i)}{w}\right) \\
&= \frac{x}{w} + \exp\left(\frac{-(t-u)}{w}\right) \tilde{F}(u). \qquad (1)
\end{aligned}$$

4.2 Combining Efficient Features in a Framework

While EDWTs are efficiently computable, they are weaker predictors of popularity than self-exciting processes with more complex kernels as shown in our evaluation (§6). We overcome this limitation of EDWTs with the second key insight in the Chess design: combining many weak, but readily computable, signals through a learning framework achieves high accuracy while remaining efficient. We use a neural network as our learning framework with two types of features as input: stateless and stateful.

Stateless features are quantities that do not change dramatically during the life-cycle of a video. A prediction service does not need to keep any state associated with these features or their past values. Instead it can query

them from the social network at prediction time. For our purposes, the most important are the social features, including the number of likes and friends of the video owner. They also include the video's length, its age, and several other easily queryable social features.

Stateful features are quantities that can vary dynamically throughout the life-cycle of the video. Past access patterns are one type of stateful feature. The changing pattern of the number of comments, likes, shares, saves for later viewing, etc. are all stateful features as well. They are stateful in that a prediction service needs to keep state associated with them between predictions. We use exponential kernels to keep this state constant per-video and we combine four kernels with different time windows—1, 4, 16, and 64 hours—to capture more complex patterns.

We use the stateless and stateful features as input to a 2-layer neural network (NN) with 100 hidden nodes for predicting total future watch time. We find that neural networks reduce the prediction error by 40% compared to linear models, but more complex models, i.e., adding more layers or using more hidden nodes do not further improve accuracy. We initially selected all features from the social network that we thought could provide some signal and then trimmed those that did not have an effect on prediction accuracy. We made features stateless or stateful based on our intuition, e.g., friends of the video owner is stateless because it changes little during the lifetime of the video. We also tried several different sets of time windows for stateful features and settled on 1, 4, 16, and 64 hours as providing the highest accuracy. We did this feature engineering using a setup similar to the single prediction experiments in our evaluation, on a separate and earlier month-long trace.

Another important technique for boosting accuracy is logarithmic scaling of both the feature values and prediction targets. Because these values can vary from 10-10^8 depending on video popularity, they need to be properly scaled to avoid optimization difficulties. Although linear scaling, in the form of standardization [6], is the commonly used method in statistical learning, we find that *logarithmic scaling*, i.e., $x \rightarrow \log(x+1)$, delivers much better performance for our workload. It ensures the model is not biased towards only predicting extremely popular videos, achieves good prediction accuracy across the whole popularity spectrum, and improves the coverage ratio of QuickFire by as much as 6% over linear scaling. We use this method in all our evaluations.

4.3 Efficient Online Model Update

Naively training our model would require a large set of training examples with their full future watch time, which is unknown. To address this issue, we use an *example queue* to generate training examples from the recent past, and use them as approximations for the future. When a video is accessed, its current state is appended to the queue. While the video is in the queue we track its watch time and feature values. Later, when an example is evicted from the queue it becomes training data with the difference in watch time between its entry and eviction used as the target future watch time. As an added benefit, because examples keep entering and being evicted from the queue, the prediction model is continuously updated at a constant learning rate to keep up with changes in the workload.

The example queue needs to be carefully designed in order to minimize the memory and CPU overhead while achieving the best model accuracy. We found that two design parameters are key to balancing this trade-off: prediction horizon and example distance. Section 6.3 investigates the effect of varying each parameter and shows that setting them properly leads to high accuracy with low memory and CPU overhead.

The *prediction horizon* is the time difference between entry and eviction of examples from the queue. In other words, an example is evicted and becomes training data when its age in the queue exceeds the prediction horizon. A larger horizon provides a better approximation of total future watch time, but it also results in a longer queue with higher memory usage. For our workload, a prediction horizon of 6 days achieves a good tradeoff with high accuracy and low overhead.

We found our example queue was flooded by data points from the most popular videos due to the skewed power law distribution in video access. Many of these data points were effectively redundant and did not help improve accuracy. This is because the input values and the prediction target will be very similar for the same video at two nearby time points. We skip these redundant examples using an admission policy that only allows a new example into the queue if the difference between its timestamp and the most recent example for the same video is greater than a threshold. We call this threshold the *example distance* D because it ensures there is at least D time between all examples of the same video. Although this alters the training data distribution, we find $D = 2\text{h}$ achieves high accuracy while greatly reducing memory overhead, due to the high skew and large volume of data.

5 The Implementation of ChessVPS

To make video popularity predictions continuously available we implemented the Chess video prediction service (ChessVPS). ChessVPS validates the scalability of our design by providing popularity prediction for Facebook's video workload while running on only four machines.

Figure 3 provides a high-level view of the architecture of ChessVPS. The service uses 8 workers distributed across 4 machines to generate predictions on the full workload. The key steps in the process are: 1) ingesting access logs,

2) querying for additional features, 3) making predictions, 4) serving predictions, and 5) updating the models.

Ingesting Access Logs Video accesses on Facebook are logged to Scribe [15]. We ingest the access logs by continuously streaming them from Scribe. To handle this streaming load—as well as distribute prediction computation—we use 8 worker processes on 4 machines. The access logs in Scribe are sharded based on video ID and each worker streams one-eighth of the shards.

Querying for Additional Features Each worker augments the access logs with the additional features it queries from TAO [7], Facebook's cache for the social graph. Current values of these features are already stored in TAO, e.g., the number of likes of a video is stored in TAO so it can be presented along with the video. Stateless features are directly added to each access of a video. The 4 exponentially decayed counters for each stateful features, with varying time windows, are updated upon every access, and the values added to the feature set as well.

We reduce the overhead from querying for additional features in three ways. First, we batch queries and only dispatch them once we have ingested 1000 accesses. Second, we deduplicate queries for the same video in a batch. Third, we cache results from TAO for 10 minutes, which reduces the load we impose on TAO by over 50%.

Making and Serving Predictions Each worker maintains a table with its most recent predictions for the top 10 million most popular videos in its shard. The 80 million videos in all shards encompass the actively accessed video working set on Facebook. After the worker queries TAO for additional features, it updates the exponentially decayed kernels, and feeds all feature values into the neural network to calculate a prediction—the design of Chess has enabled us to do all this in real-time, on a small set of machines. This prediction is then used to update the video's entry in the table. Every 10 minutes each worker scans its table and sorts videos based on their predictions. An aggregator in each machine collects the top 1 million videos from the collocated workers, and then it broadcasts its predictions to all aggregators and waits for their predictions. Once an aggregator has the top 1 million predictions from all 8 workers, it merges and sorts them. It then caches the aggregated predictions and uses them to answer requests for the next 10 minutes. Other services, e.g., a re-encoding service, can query any worker to learn the videos that we predict to be the most popular.

Updating the Model and Memory Overhead To reduce space overhead we maintain one model and example queue per machine (shared between two workers). We use an example queue with a prediction horizon of 6 days and an example distance of 2 hours to keep the memory overhead low. We further reduce the memory overhead of the example queue by only admitting a consistently sampled 30% of the videos to it—this proportionally reduces the queue size, without causing any model overfitting.[2] The resulting example queue consumes ~6 GB of memory per machine, or ~24 GB in total.

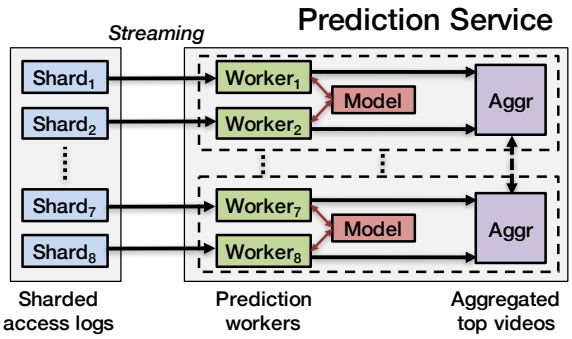


Figure 3: Chess video prediction service architecture.

Each video has 12 stateless features and 7 stateful features. These features, associated metadata, and current popularity prediction add up to a storage overhead of ~250 bytes per video. Thus, all 80 million videos use ~20GB RAM in total to maintain. This results in a total memory overhead of ~44GB RAM from models and metadata, or only ~11GB RAM per machine. In contrast, if SEISMIC were used instead, the timestamp and watch time of each past request would need to be stored to make predictions, translating to 1.2MB per video on average and ~10TB total memory usage.

6 Evaluation

Our evaluation seeks to answer three key questions for Facebook's video workload:

1. How does the prediction accuracy of Chess compare to the heuristic used in production, the state of the art, and a clairvoyant predictor?
2. What are the effects of our design decisions, such as prediction target scaling, prediction horizon, and example distance, on accuracy and resource usage?
3. How would adopting ChessVPS for production processing decisions impact resource consumption and watch time coverage?

6.1 Experimental Setup

Predictors Table 1 shows the predictors we compare in this evaluation in three groups: baselines, increasing subsets of Chess, and a clairvoyant predictor. Among the baselines, we modified SEISMIC and Initial(1d) to suit our application scenario better, and tuned their parameters

[2] This sampling turned out to be unnecessary, as even without it the memory footprint per machine is still only 25GB.

Predictor	Ranking of videos based on:
Initial(1d) [36]	Watch time in the initial day after upload, or total watch time if less than a day old.
SEISMIC-CF [45]	State of the art popularity prediction using a power-law kernel, with followers of each viewer set to constant for our application.
Owner-Likes	Like count of the video owner. This was recently used in production.
EDWT(4h)	Exponentially decayed watch time with a four hour time window.
NN(EDWT)	Neural network model using only EDWT features with time windows 1h, 4h, 16h, 64h.
CHESS	Neural network model with stateless features (e.g., owner likes) and stateful features (e.g., video views, video likes) made efficient using EDWTs.
Clairvoyant	Total future watch time of each video. This is unattainable in practice.

Table 1: Popularity predictors evaluated on Facebook's video workload in our evaluation.

to yield the best performance on our dataset. The original SEISMIC algorithm needs the number of followers of each retweeter for predicting tweet popularity, which is unsuitable for video watch time prediction on Facebook because a viewer might not share the video after watching and directly influence its followers. Based on a parameter sweep, we settled on a constant 1000 for this setting on our workload, with the ensuing method called SEISMIC-CF—as shown below, its performance remains competitive even with this modification. Initial(1d) [36] originally uses the number of requests—watch time in our case—of the entire first day for predicting popularity, but for our application, if the video is less than 1 day old we use its total watch time to generate a prediction instead of waiting.

Comparing to baselines that represent the state of the art—Initial(1d) and SEISMIC-CF—and a recent production heuristic—Owner-Likes—enables us to quantify how much CHESS improves on the state of the art and would improve production. Comparing increasing subsets of CHESS—EDWT(4h) and NN(EDWT)—allows us to quantify the improvement from each addition to CHESS. Comparing to a clairvoyant predictor allows us to quantify how far CHESS is from a perfect predictor.

Experimental Methods and Workloads We use three experimental methods with progressively more realistic results and time-consuming experiments: single prediction, simulation, and real-time sampled processing. The single prediction method resembles that used by prior work on popularity prediction [18, 45] and enables comparisons with SEISMIC. The simulation method enables us to run many experiments in a reasonable time frame and we validate its results using real-time sampled processing.

Workloads. Single prediction and simulation experiments each use the same 35-day trace of video access as their workload. The trace is comprised of full access logs for a random sample of 1% of videos during 5 weeks. The workload for the real-time sampled processing experiments was the full Facebook video workload.

Single prediction. The memory and computational overhead of SEISMIC[3] made it infeasible for us to run the more realistic simulation (or real-time sampled processing) experiments with it, so we designed the single prediction method to enable evaluation against it. In this method each predictor takes as input the historical information for a video up to a time point and then issues predictions. The predictions are then evaluated using the watch time of the video in the 15 days immediately following the time point.

The input historical information and future watch time of the videos are extracted from the trace as follows. First, we select only the videos in the trace that are accessed on one day at the midpoint of the trace. This limits the size of the prediction to make the experiments feasible. Second, we randomly pick a time point on that day for each video to control for diurnal effects. Finally, we extract the trace up to the time point for each video and the future watch time in the 15 days following the time point.

Simulation. Our main evaluation method is simulation of a video prediction service that runs hourly using our 35-day trace. In each simulation, we replay the whole trace, train our prediction model continuously, and the predictor ranks videos for re-encoding every hour. Once a video is selected for re-encoding, it is recorded in a hash table. The hash table is then queried for each request to see whether the requested video has already been re-encoded before. We use the initial 23 days of the trace to populate the hash table, and report results on the last 12 days.

Real-time sampled processing. Our final evaluation method is the most realistic and follows the description in Section 5. The whole service operates on 4 machines, each with 20 2.8GHz cores and 32GB memory, and processes access logs of all Facebook videos in real time. We then write a client using results from CHESSVPS to make encoding decisions in 10 minute intervals. The whole system was run for a week for warm up and we present the results from the next day.

[3]The implementation of SEISMIC is ~200× slower than CHESS's implementation. However, part of this slowdown stems from SEISMIC being implemented in the R language [45].

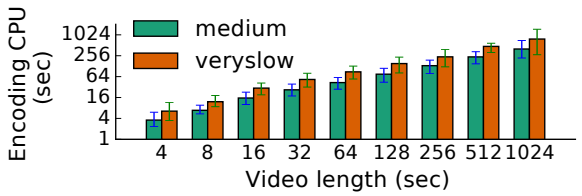


Figure 4: The linear relationship between video length and encoding CPU makes video length a reasonable proxy for encoding CPU.

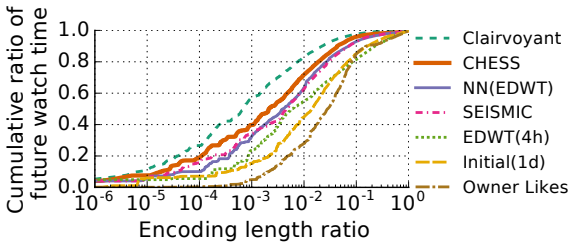


Figure 5: Single prediction results shown by the cumulative ratio of future watch time of the videos selected by each predictor for a given encoding length ratio.

Metrics Our ideal metrics for evaluating predictors would include the future watch time ratio of re-encoded videos and the encoding overhead from doing additional processing on them. Neither of these metrics is feasible for us to collect, but we can gather reasonable approximations of them nevertheless. In the prediction experiments, total future watch time is impossible to collect because there is always more future. Instead we track watch time within a 15-day period because popularity of Facebook videos typically stabilizes in one week (from Figure 2b). In simulations and real-time processing, we keep track of the watch time coverage of re-encoded videos in every hour, and find the coverage ratio stabilizes within 5 days after enough recently popular videos have been re-encoded, so in simulations we have a 23 day warm-up period and report the average coverage ratio in the next 12 days trace, while in real-time processing we wait 1 week before reporting results in the next day. Doing additional processing on all videos is not feasible because it would require the use of a fleet of machines much larger than the current processing fleet. Instead we approximate processing overhead using video length and by doing sampled processing.

Video length is a reasonable proxy for processing CPU. We use video length as our overhead metric for single prediction and simulation experiments because it is fast to compute and a reasonable proxy for processing CPU usage. To demonstrate it is a reasonable proxy we randomly sampled 3000 videos uploaded to Facebook, bucket them by $\log_2$ of their lengths, and show the 20th percentile, median, and 80th percentile CPU usage for FFmpeg "medium" and "veryslow" encodings in each bucket. While there is a large variance in each bucket, the CPU usage is approximately linear in the video length. Statistically this is a strong linear relationship with $R^2 = 0.981$ between length and median CPU usage across the buckets. Based on this observation, in both single prediction and simulation experiments, we rank the videos with each method, and re-encode the top videos until the total length exceeds a threshold (representing a fixed CPU budget). We then compute its ratio to total length of all videos, terming the quantity "encoding length ratio".

Sampled processing. We use measured CPU usage from processing a sample of videos as our overhead metric for the real-time sampled processing experiment. For this experiment, a 0.5% random sample of the selected videos for each predictor ($\approx$ 3000 in total) are re-encoded using QuickFire. At the same time, 5000 videos are sampled from the video uploads that day and encoded with FFmpeg "veryslow". We then calculate the overhead for encoding the selected videos using the measured encoding time for these two sets: let U denote the average FFmpeg "veryslow" encoding time of the sampled video uploads, and Q the average QuickFire encoding time of the videos selected by one method in the sample set, with 95% confidence interval $[Q^-, Q^+]$ (computed using `scikits-bootstrap` [14]). If N is the total number of videos selected by that method, and M the daily video uploads to Facebook, then we estimate the CPU overhead to be $\frac{QN}{UM}$, with confidence interval $\left[\frac{Q^-N}{UM}, \frac{Q^+N}{UM}\right]$, which helps us estimate the variance from sampled processing.

6.2 Single Prediction Experiments

The results of the single prediction experiment that enable us to compare to SEISMIC are shown in Figure 5. The results generally follow the intuition that predictors with more information available to them will make better predictions. For instance, Initial(1d) and Owner-Likes each perform poorly because they use only a single scalar value as their prediction. We highlight two results.

NN(EDWT) is competitive with SEISMIC-CF. EDWT(4h) is a self-exciting process prediction method inspired by SEISMIC with the primary difference being the use of an exponentially decayed kernel that makes it much more resource efficient. The gain in resource efficiency, however, comes with a consistently lower coverage ratio for EDWT(4h) than for SEISMIC-CF. For instance, to achieve 80% coverage EDWT(4h) needs to select 2.9× more minutes of video than SEISMIC-CF.

NN(EDWT) is a combination of four EDWTs in a neural network model. It performs slightly worse (up to 6% lower watch time coverage) than SEISMIC-CF when encoding a very small fraction of videos ($< 0.1\%$). When encoding a more typical fraction of videos ($> 0.1\%$), however,

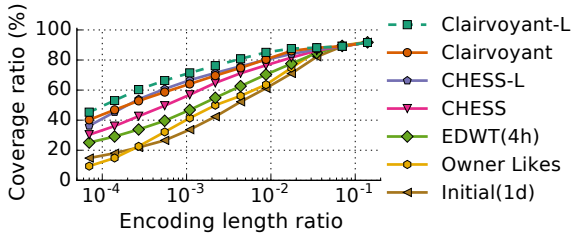


Figure 6: Simulation results shown by the watch time coverage ratio of videos selected by each predictor for a given encoding length ratio. Clairvoyant-L/Chess-L denotes the corresponding algorithm with scores normalized by video length.

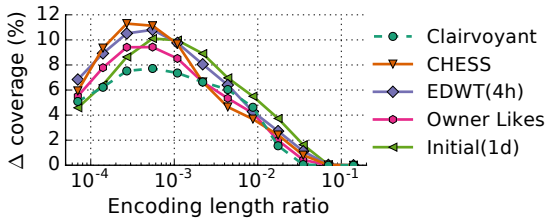


Figure 7: Improvement of coverage ratio through score normalization by video length.

it achieves similar or slightly higher performance than SEISMIC-CF. Both of these methods are based solely on past access patterns, which indicates our learning framework is able to deliver comparable results to a handcrafted algorithm even when only using features of lesser quality and consuming fewer resources.

Chess provides higher accuracy. The full Chess provides the highest watch time coverage of all achievable predictors we evaluated and is the closest to the clairvoyant predictor. Its improvement over SEISMIC-CF is significant: it achieves 40% watch time coverage with 2.0× fewer minutes of video, 60% coverage with 1.8× fewer minutes, and 80% coverage with 1.6× fewer minutes.

6.3 Simulation Experiments

We used simulation experiments to provide a more realistic comparison to other predictors and to investigate the effects of three design parameters: prediction target scaling, prediction horizon, and example distance.

Chess provides higher accuracy. Figure 6 show the watch time coverage of all predictors except SEISMIC-CF which is excluded because of its high memory usage and slow speed. The relative performance of different methods are similar to the single prediction experiment (Figure 5), with Chess and Chess-L outperforming other practical methods, which validates those results. For instance, to reach 80% coverage, Chess-L encodes 2× as many video-minutes as Clairvoyant-L, while Owner-Likes encodes 8×. The overall performance at lower encoding length ratios ($<10^{-3}$), however, improves for two reasons: (1) due to the power-law distribution of popularity, the simulation will include a larger number of the most popular videos than the single prediction experiments that use a smaller sample, (2) in simulations a video is likely re-encoded shortly after gaining popularity, therefore covering more watch time, whereas in the single prediction experiment a random time point is picked to divide the past and future. The second reason also explains why Owner-Likes now outperforms Initial(1d) under many settings even though it did worse in the single prediction experiments. With Owner-Likes, videos are re-encoded at upload time, and so the benefits of re-encoding start accumulating immediately. In contrast, Initial(1d) always waits up to 1 day until a video is popular to select it and misses many of its early views. For most settings, the benefit from higher accuracy in Initial(1d) does not make up for the early views it misses relative to Owner-Likes.

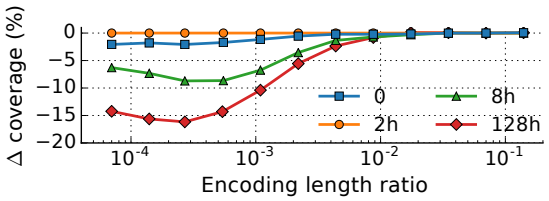
The coverage ratio of different results saturates and converges to the same value when encoding length ratio is above 7%, but because QuickFire takes 20× CPU. This translates to 140% additional CPU usage, a big increase to the already large fleet.

Score normalization by length improves accuracy. Figure 7 shows the increase in coverage of each method with and without score normalization by video length. We find this technique consistently improves the performance of all methods, with Chess seeing the biggest improvement and Clairvoyant seeing the smallest, which reduces the gap between the two.

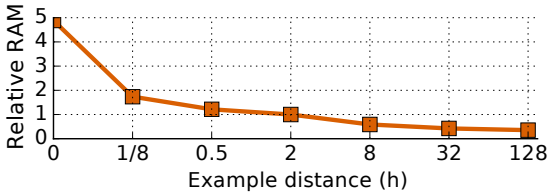
Clairvoyant-L and Chess-L in Figure 6 show the two corresponding methods with scores normalized by video length. Clairvoyant-L achieves the highest coverage ratio consistent with Section 3.1: 80%+ with 1% encoding length ratio, and 70%+ with 0.1% encoding length ratio. Chess-L delivers the best result among all the practical methods, only 6%–8% lower than Clairvoyant-L.

Chess-L improves the coverage ratio of the production baseline, Owner-Likes by 8%-30% with the same encoding length ratio from 0.01%-2%. To achieve 80% coverage ratio, Chess-L only needs to encode 0.9% of total video length, while Owner-Likes needs 2.5%. The results are especially favorable at the middle to lower end of the encoding length ratio. We hope this result motivates the design of new encoding algorithms that utilizes even higher CPU usage to achieves even better compression ratios. Even if this encoding method uses 100× the CPU of FFmpeg, with Chess-L, 64% of the watch time can still be served with only ~10% CPU overhead from re-encoding 0.1% of videos.

Increasing the prediction horizon has diminishing returns and higher memory usage. Due to space limi-



(a) Example distance vs. relative coverage difference.



(b) Memory usage with different example distances.

Figure 8: Effects of example distance.

tations we only summarize the results from varying the prediction horizon. We experiment with horizons of 1h, 1d, 2d, 4d, 6d, 8d, 12d, and `inf`. The coverage is lowest when prediction horizon is as short as 1 hour. It then improves as the prediction horizon increases until 6d, then however, when the horizon is 8d and `inf`, the coverage drops by 1%-2%. Because the training target of our model is the watch time within the prediction horizon, a longer horizon means a better approximation for total future watch time and improves the result. However, when the horizon is too long, the training examples evicted from the queue were created a long time ago, and the stale training data hurts the prediction accuracy.

Meanwhile, the memory usage of the queue grows roughly linearly with the prediction horizon because examples within the horizon are held in the queue. As it provides the highest coverage with modest memory usage in ChessVPS, we choose 6d to be the default prediction horizon in our evaluations.

A short example distance increases coverage and decreases memory usage. Example distance, the minimum time distance between two examples of the same video, is another knob controlling the trade-off between coverage and system resource usage. We have run experiments with values 0, $\frac{1}{8}$h, $\frac{1}{2}$ h, 2h, . . . , 128h, and show the relative coverage compared to the default 2h in Figure 8a, and the simulator's relative memory usage in Figure 8b. We have omitted some lines in the former for clarity but describe the results below.

The memory usage of the queue drops monotonically as the example distance D increases. When D = 2h, we reduce memory by 5× compared to D = 0 (not using the heuristic) because most examples from the popular videos never enter the queue. Interestingly, the coverage ratio increases a little at the same time because the examples

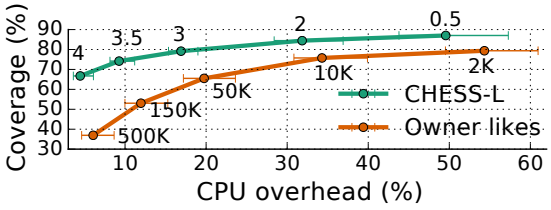


Figure 9: Projected impact of Chess-L compared to Owner-Likes. Encoding score thresholds are annotated for each data point.

skipped are all "duplicates" of the most popular videos; removing them has little effects on training set diversity while making the model less biased towards those videos. This improves the overall performance similar to the effects of *logarithmic scaling*. If D further increases, memory usage continues to drop, though at the expense of the much lower coverage ratio, up to ~15% at 128h. Under such a setting, most examples from even the moderately popular videos are filtered out, and the model fails to deliver accurate predictions. Based on these results we have picked 2h as our default example distance.

6.4 Real-time Sampled Processing

We validate our algorithm and implementation by deploying ChessVPS and running it in real-time with the production access logs. Although the real-world encoding logic is complex and our service is not used by the production encoding pipeline yet, we have implemented a "pseudo client" that queries the service every 10 minutes and issues encoding decisions based on the prediction scores. This way we can monitor the coverage and encoding statistics in real time, and verify its projected impact more realistically. In simulations we ranked the videos every hour and encoded them until the total video lengths reach a threshold, but to more closely resemble the production heuristic here, which re-encodes videos whose owners have more than 10K likes, we also issue re-encoding decisions for videos with prediction scores exceeding a threshold. In the following discussion Chess(α) means Chess with score threshold α, and similarly for Owner-Likes(β).

Figure 9 shows the real-time sampled processing results. Chess-L(3) achieves ~80% coverage ratio as Owner-Likes(10K), while reducing the re-encoding CPU overhead from 54% to 17%. At slightly lower CPU usage, Chess-L(2) improves the coverage of Owner-Likes(10K) from 75.7% to 84.4%. The improvement of Chess-L is greater at lower CPU overhead settings. For example, Owner-Likes(500K) delivers 37% coverage with 6% CPU overhead, whereas Chess-L(4) achieves 66.7% coverage with 4.5% CPU overhead. This is favorable for limited computing budgets, or if we want to apply even more

computing intensive encoding methods or have more encoded versions. The relative performance between the two methods concords with the simulation results shown in Figure 6; the minute differences stem from a changing workload and the logic for different encoding thresholds.

7 Related Work

Our work explores building a scalable and accurate popular video prediction service, with applications on re-encoding for improving streaming quality. In this section we discuss related work on popularity prediction, video quality of experience (QoE) optimization, and caching, which we draw inspiration from for this study.

Popularity Prediction In recent years, the popularity prediction of online content has attracted intense research attention. Simple heuristics like counting requests in the first few hours/days [36], or followers of the owner are fast but inaccurate. Meanwhile, various methods have been proposed for modeling Twitter/Facebook resharing [10, 9, 44]. They usually maximize accuracy, rely on more features and are memory/computation intensive, e.g., requiring to store and scan multiple features of each retweet/sharing when making every prediction. Our method is designed for both accuracy and efficiency, and delivers accurate, real-time prediction for all Facebook videos with a small hardware footprint.

Self-exciting processes have been used for modeling earthquakes [20], YouTube video accesses [13], and Twitter resharing [45]. These methods use variants of power-law kernels and thus store and process all past requests. Instead, we use an exponential kernel to cut per-video memory/computation overhead to $O(1)$. Exponentially decayed metrics are used in other contexts [12, 21]; our contribution is using them for self-exciting processes and appling them to popularity prediction. Furthermore, we are the first to combine multiple exponentially-decayed kernels in a learning framework, which allows us to match the accuracy of a power-law kernel while remaining resource efficient, thus obtaining the best of both worlds.

Video QoE Optimization As videos gain increasing importance in people's online activities, research on improving video streaming QoE has flourished. Many of them focus on the delivery path, e.g., selecting the best bit-rate per chunk in ABR for efficiency, stability and fairness [23, 24, 25, 42], and building a control plane for video delivery [17, 28, 30]. On the upload and encoding path, video codecs have evolved towards using higher computation in exchange for higher compression, from MPEG-2 [19] to the now widely adopted H.264 [34], and gradually moving to the next generation codecs such as VP9 [31] and H.265 [35]. In addition, QuickFire [1, 41] and Netflix per-title encoding [4] try to improve compression of existing codecs by finding the best encoding configuration based on video content as well as resolution. We explore another dimension in video encoding based on feedback from delivery. By applying more processing to popular videos, we optimize the overall trade-off between encoding CPU and streaming QoE.

Caching We find the video re-encoding problem also bears some interesting similarities to caching. By locating hot data in a small but fast storage, caching saves access latency and bandwidth [37]. Meanwhile, by spending more CPU on the popular videos, re-encoding improves the video streaming quality at given network conditions.

Many caching algorithms have been designed to exploit different characteristics of request patterns, including recency (LRU [26]), frequency (LFU [29]), or both (SLRU [27], MQ [46]). The exponentially decayed kernel used as a building block in Chess combines both recency and frequency, and the trade-off is tuned through the time window parameter. Similar to length normalization, size-aware caching [8, 11] also favors smaller items so more can be cached in limited space, improving object hit-ratio.

8 Conclusion

Facebook serves billions of videos views every day and new videos are uploaded at a rapid rate. With limited CPU resources, it is challenging to identify which of these videos would most benefit from re-encoding with computing intensive methods like QuickFire that enhance the viewing experience.

We have described an efficient video popularity prediction service that has the Chess algorithm at its core. Chess achieves scalability by summarizing past access patterns with a constant number of values, and it achieves efficiency by combining the past access patterns and other features in a continuously updated neural network model. Our evaluation show that compared to a recent production heuristic, Chess reduces encoding CPU required by 3× to cover 80% of user watch time with QuickFire.

While the focus of this paper has been popularity prediction for the Facebook video workload, we conjecture that our ChessVPS approach would generalize to efficiently predict popularity in other settings.

Acknowledgments We are grateful to our shepherd Vishakha Gupta-Cledat, the anonymous reviewers of the ATC program committee, Siddhartha Sen, Haoyu Zhang, Theano Stavrinos, and Aqib Nisar for their extensive comments that substantially improved this work. We are also grateful to Sergiy Bilobrov, Minchuan Chen, Maksim Khadkevich, and other Facebook engineers for their discussion on this problem. Our work is supported by Facebook, NSF CAREER award #1553579, and a Princeton University fellowship.

References

[1] QuickFire technology explained @Scale. https://www.facebook.com/atscaleevents/videos/1682906415315789.

[2] Facebook Community Update. https://www.facebook.com/photo.php?fbid=10102457977071041.

[3] Facebook's Streaming Video Engine @Scale Talk. https://www.facebook.com/atscaleevents/videos/1741710496102047/.

[4] A. Aaron, Z. Li, M. Manohara, J. De Cock, and D. Ronca. Per-Title Encode Optimization. http://techblog.netflix.com/2015/12/per-title-encode-optimization.html.

[5] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel, et al. Finding a needle in haystack: Facebook's photo storage. In *USENIX OSDI*, 2010.

[6] C. M. Bishop. Pattern recognition. *Machine Learning*, 128, 2006.

[7] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, et al. Tao: Facebook's distributed data store for the social graph. In *USENIX ATC*, 2013.

[8] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *USITS*, 1997.

[9] G. H. Chen, S. Nikolov, and D. Shah. A latent source model for nonparametric time series classification. In *ACM NIPS*, 2013.

[10] J. Cheng, L. Adamic, P. A. Dow, J. M. Kleinberg, and J. Leskovec. Can cascades be predicted? In *ACM WWW*, 2014.

[11] L. Cherkasova. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Hewlett-Packard Laboratories, 1998.

[12] G. Cormode, F. Korn, and S. Tirthapura. Exponentially decayed aggregates on data streams. In *IEEE ICDE*, 2008.

[13] R. Crane and D. Sornette. Robust dynamic classes revealed by measuring the response function of a social system. *PNAS*, 2008.

[14] C. Evans. scikits-bootstrap. https://github.com/cgevans/scikits-bootstrap.

[15] Facebook. Facebok Scribe. https://github.com/facebook/scribe/wiki.

[16] FFmpeg. The FFmpeg project. http://ffmpeg.org.

[17] A. Ganjam, F. Siddiqui, J. Zhan, X. Liu, I. Stoica, J. Jiang, V. Sekar, and H. Zhang. C3: Internet-scale control plane for video quality optimization. In *USENIX NSDI*, 2015.

[18] G. Gürsun, M. Crovella, and I. Matta. Describing and forecasting video access patterns. In *INFOCOM, 2011 Proceedings IEEE*, pages 16–20. IEEE, 2011.

[19] B. G. Haskell, A. Puri, and A. N. Netravali. *Digital Video: An Introduction to MPEG-2*. Springer Science & Business Media, 1997.

[20] A. G. Hawkes. Spectra of some self-exciting and mutually exciting point processes. *Biometrika*, 1971.

[21] J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang. Efficient identification of hot data for flash memory storage systems. *ACM TOS*, 2006.

[22] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of facebook photo caching. In *ACM SOSP*, 2013.

[23] T.-Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. *ACM SIGCOMM Computer Communication Review*, 2015.

[24] J. Jiang, V. Sekar, and H. Zhang. Improving fairness, efficiency, and stability in HTTP-based adaptive video streaming with festive. In *ACM CoNEXT*, 2012.

[25] J. Jiang, V. Sekar, H. Milner, D. Shepherd, I. Stoica, and H. Zhang. CFA: A practical prediction system for video QoE optimization. In *USENIX NSDI*, 2016.

[26] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. 1994.

[27] R. Karedla, J. S. Love, and B. G. Wherry. Caching strategies to improve disk system performance. *IEEE Computer*, 1994.

[28] X. Liu, F. Dobrian, H. Milner, J. Jiang, V. Sekar, I. Stoica, and H. Zhang. A case for a coordinated internet video control plane. In *ACM SIGCOMM*, 2012.

[29] S. Maffeis. Cache management algorithms for flexible filesystems. *ACM SIGMETRICS Performance Evaluation Review*, 1993.

[30] M. K. Mukerjee, D. Naylor, J. Jiang, D. Han, S. Seshan, and H. Zhang. Practical, real-time centralized control for cdn-based live video delivery. *ACM SIGCOMM Computer Communication Review*, 2015.

[31] D. Mukherjee, J. Bankoski, A. Grange, J. Han, J. Koleszar, P. Wilkins, Y. Xu, and R. Bultje. The latest open-source video codec VP9-an overview and preliminary results. In *IEEE PCS*, 2013.

[32] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al. F4: Facebook's warm blob storage system. In *USENIX OSDI*, 2014.

[33] I. Sodagar. The MPEG-DASH standard for multimedia streaming over the internet. *IEEE MultiMedia*, 2011.

[34] G. J. Sullivan, P. N. Topiwala, and A. Luthra. The h. 264/avc advanced video coding standard: Overview and introduction to the fidelity range extensions. In *SPIE Optics + Photonics*, 2004.

[35] G. J. Sullivan, J.-R. Ohm, W.-J. Han, and T. Wiegand. Overview of the high efficiency video coding (HEVC) standard. *IEEE CSVT*, 2012.

[36] G. Szabo and B. A. Huberman. Predicting the popularity of online content. *CACM*, 2010.

[37] A. S. Tanenbaum and A. S. Woodhull. *Operating systems: design and implementation*, volume 2. Prentice-Hall Englewood Cliffs, NJ, 1987.

[38] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. RIPQ: advanced photo caching on flash for facebook. In *USENIX FAST*, 2015.

[39] T. C. Thang, Q.-D. Ho, J. W. Kang, and A. T. Pham. Adaptive streaming of audiovisual content using MPEG DASH. *IEEE Transactions on Consumer Electronics*, 2012.

[40] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the h. 264/avc video coding standard. *IEEE Transactions on circuits and systems for video technology*, 2003.

[41] WIRED. Facebook acquires QuickFire Networks. https://www.facebook.com/wired/posts/10152676478868721.

[42] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over HTTP. *ACM SIGCOMM Computer Communication Review*, 2015.

[43] T. Zaman, E. B. Fox, E. T. Bradlow, et al. A Bayesian approach for predicting the popularity of tweets. *The Annals of Applied Statistics*, 2014.

[44] T. R. Zaman, R. Herbrich, J. Van Gael, and D. Stern. Predicting information spreading in Twitter. In *ACM NIPS Workshop on computational social science and the wisdom of crowds*, 2010.

[45] Q. Zhao, M. A. Erdogdu, H. Y. He, A. Rajaraman, and J. Leskovec. SEISMIC: A self-exciting point process model for predicting tweet popularity. In *ACM SIGKDD*, 2015.

[46] Y. Zhou, J. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *USENIX ATC*, 2001.

Squeezing out All the Value of Loaded Data: An Out-of-core Graph Processing System with Reduced Disk I/O

[1]Zhiyuan Ai* [1]Mingxing Zhang* [1]Yongwei Wu [2]Xuehai Qian [1]Kang Chen [1]Weimin Zheng
[1]*Tsinghua University*† [2]*University of Southern California*

Abstract

The current primary concern of out-of-core graph processing systems is improving disk I/O locality, which leads to certain restrictions on their programming and execution models. Although improving the locality, these constraints also restrict the expressiveness. As a result, only sub-optimal algorithms are supported for many kinds of applications. When compared with the optimal algorithms, these supported algorithms typically incur *sequential*, but *much larger*, amount of disk I/O.

In this paper, we explore a fundamentally different tradeoff: less total amount of I/O rather than better locality. We show that out-of-core graph processing systems uniquely provide the opportunities to lift the restrictions of the programming and execution model (e.g., process each loaded block at most once, neighborhood constraint) in a feasible manner, which enable efficient algorithms that require drastically less number of iterations. To demonstrate the ideas, we build CLIP, a novel out-of-core graph processing system designed with the principle of "squeezing out all the value of loaded data". With the more expressive programming model and more flexible execution, CLIP enables more efficient algorithms that require much less amount of total disk I/O. Our experiments show that the algorithms that can be only implemented in CLIP are much faster than the original disk-locality-optimized algorithms in many real-world cases (up to tens or even thousands of times speedup).

1 Introduction

As an alternative to distributed graph processing, disk-based single-machine graph processing systems (out-of-core systems) can largely eliminate all the challenges of using a distributed framework. These systems keep only a small portion of active graph data in memory and spill the remainder to disks, so that a single-machine can still process large graphs with the limited amount of memory. Due to the ease of use, several out-of-core systems have been developed recently [15, 26, 41]. These systems make practical large-scale graph processing available to anyone with a modern PC. It is also demonstrated that the performance of a single ordinary PC running GridGraph is competitive with a distributed graph processing framework using hundreds of cores [41].

The major performance bottleneck of out-of-core systems is disk I/O. Therefore, improving the locality of disk I/O has been the main optimization goal. The current systems [15, 26, 41] use two requirements to achieve this goal. First, the execution engine defines a specific processing order for the graph data and only iterates the edges/vertices according to such order, which means that each edge/vertex is processed *at most once* in an iteration. By avoiding fully asynchronous execution, this technique naturally reduces the tremendous amount of random disk I/O that would have otherwise occurred. The second is the *neighborhood constraint* that requires a single user-defined programming kernel to access only the neighborhood of its corresponding input vertex/edge. This requirement improves the locality of disk I/O and also makes automatic parallelization of in-memory processing practical.

According to our investigation, almost all existing out-of-core systems enforce the above two requirements in their programming and execution models, which assure the good disk I/O locality for the algorithms that they supported. However, these restrictions (e.g., process each loaded block at most once, neighborhood constraint) also affect the models' expressiveness and flexibility and lead to the sub-optimal algorithms. As a result, the execution incurs *sequential, but excessive*, the amount of disk I/O, compared with more efficient algorithms which require drastically less iterations.

As an illustration, the "at most once" requirement obviously wastes the precious disk bandwidth. Many graph algorithms (e.g. SSSP, BFS) are based on iterative improvement methods and can benefit from iterating multiple times on a loaded data block. Moreover, many important graph problems (e.g., WCC, MIS) can be solved with much less iterations (typically only **one** pass is enough) by changing algorithms. However, these algorithms require the removal of "neighborhood constraint". In essence, we argue that the current systems follow a wrong trade-off: they improve the disk I/O locality at the expense of less efficient algorithms with the larger amount of disk I/O, wasting the precious disk

*Z. Ai and M. Zhang equally contributed to this work.

†Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology (TNLIST), Tsinghua University, Beijing 100084, China; Research Institute of Tsinghua University in Shenzhen, Guangdong 518057, China.

bandwidth. As a consequence, current out-of-core systems only achieve *sub-optimal* performance.

In this paper, we propose CLIP, a novel disk-based graph processing system, in which supporting more efficient algorithms is the primary concern. We argue that out-of-core graph processing systems uniquely provide the opportunities to lift the restrictions of the programming and execution model (e.g., process each loaded block at most once, neighborhood constraint) in a feasible manner. Specifically, CLIP is designed with the principle of "squeezing out all the value of loaded data", It defines a programming model that supports *1)* **loaded data reentry** by allowing more flexible processing order; and *2)* **beyond-neighborhood** accesses by allowing an "edge function" to update vertex properties that do not belong to the input edge's neighborhood.

Essentially, CLIP chooses an alternative trade-off by enabling more efficient algorithms and more flexible executions at the expense of accessing vertices beyond the neighborhood. Obviously, randomly accessing vertices in disk incurs random disk I/O that is detrimental to performance. To mitigate this issue, CLIP simply *mmap* all the vertex data into memory. Without incurring development efforts, this method is vastly different from existing systems that load only needed part of vertices at a time (e.g., GraphChi, X-Stream, GridGraph).

Using this method, although the vertex data could reside in either memory or disk, Lin et al. [17] showed that the built-in caching mechanism of mmap is particularly desirable for processing real-world graphs, which often exhibit power-law degree distributions [12]. In such graphs, high-degree nodes tend to be accessed much more frequently than others and hence will always be cached in memory and result in good performance. Moreover, because the vertex data are typically much smaller than edge data but are accessed more frequently, our method is deemed to be a good heuristic in memory allocation that naturally reserves as much memory for vertices as possible. In fact, in our experiments, we *1)* test on many different real-world graphs that contain up to 6.6 billion edges; and *2)* modulate the maximum size of memory that the system is allowed to use for simulating the different size of available memory, from 32GB down to only 128MB (even 16MB for small graphs), by using cgroup. According to the results, CLIP is faster than any existing out-of-core systems on various memory limits.

The evaluation of our system consists of two parts. First, we evaluate the effectiveness of loaded data reentry, which can be applied to not only our system but also existing frameworks. According to our experiments, this simple technique can significantly reduce the number of required iterations for intrinsically iterative algorithms like SSSP and BFS, achieving up to 14.06× speedup. Second, we compare our novel beyond-neighborhood algorithms with prior ones on many important graph problems. We found that they can reduce the number of required iterations from 7∼6261 to only **one pass** for popular graph problems such as WCC (3.25×-4264× speedup) and MIS (20.9×-60× speedup).

2 Out-of-Core Graph Processing

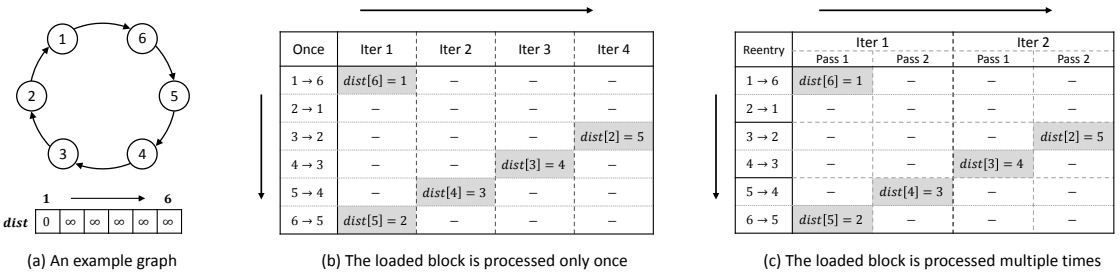
GraphChi [15] is the first large-scale out-of-core graph processing system that supports vertex programs. In GraphChi, the whole set of vertices are partitioned into "intervals", and the system only processes the related sub-graph of an interval at a time (i.e., only the edges related to vertices in this interval are accessed). This computation locality of vertex program (i.e. access only the neighborhood of input vertex) makes it easy for GraphChi to reduce random disk accesses. As a result, GraphChi requires a small number of non-sequential disk accesses and provides competitive performance compared to a distributed graph system [15].

Some successor systems (e.g., X-Stream [26], GridGraph [41]) propose an edge-centric programming model to replace the vertex-centric model used in GraphChi. A user-defined function in the edge-centric model is only allowed to access the data of an edge and the related source and destination vertices. This requirement also enforces a similar neighborhood constraint as the vertex-centric models, and hence ensures the systems to incur only limited amount of random disk I/O.

However, although these existing out-of-core graph processing systems differ vastly in detailed implementation, they share two common design patterns: *1).* Graph data (i.e. edges/vertices) is always (selectively) loaded in specific order and each of the loaded data block is processed at most *once* in an iteration; *2).* They all require that the user-defined functions should only access the *neighborhood* of the corresponding edge/vertex.

3 Reducing Disk I/O

According to our investigation, these two shared patterns could potentially prohibit programmers from constructing more efficient algorithms, and therefore increase the total amount of disk I/O. Motivated by this observation, our approach lifts the restrictions in the current programming and execution model by: *1)* providing more flexible processing order; and *2)* allowing the user-defined function to access an arbitrary vertex's property. This section discuss the rationale behind these two common patterns, and why they are *not* always necessary in an out-of-core system. More importantly, with the restrictions removed, how our approach could enable more efficient algorithms that require less number of iterations and less amount of



(a) An example graph (b) The loaded block is processed only once (c) The loaded block is processed multiple times

Figure 1: SSSP example. All the edges of this graph have the same distance set to 1.

disk I/O. In essence, our approach *squeezes out all the values of loaded data*.

3.1 Reentry of Loaded Data

Out-of-core systems typically define a specific processing order for the graph data and only iterate the edges/vertices according to such order. This is natural, because a fully asynchronous graph processing would incur the tremendous amount of *random accesses* to the graph data, drastically reducing disk I/O performance. However, this strategy could potentially increase the number of required iterations of many graph problems (e.g. SSSP, BFS) based on iterative improvement algorithms.

Figure 1 shows an example that calculates single source shortest path (SSSP) on a graph of 6 vertices. In SSSP, the vertex property $dist[v]$ is initialized to 0 for vertex 1 and ∞ for the others (Figure 1 (a)). The edge function applied to each edge (u,v) checks whether $dist[v]$ is larger than $dist[u]+1$, If it is true, $dist[v]$ is *immediately* updated as $dist[u]+1$. Figure 1 (b) shows the execution, where each iteration *sequentially* loads *one edge at a time*, processes it and updates $dist[v]$ if necessary. As a result, 4 iterations are needed. The number of iterations is determined by the diameter of the graph. To mitigate this issue, some prior systems (e.g., GraphChi, GridGraph) *1)* allows an update function to use the most recent values of the edges/vertices; and *2)* provides selective scheduling mechanisms that skip certain data blocks if they are not needed. Although these optimizations enable "asynchronous execution", the essential workflow is not changed as each block loaded is still processed *at most once* in every iteration.

We argue that the current approaches *fail to exhaust the value of loaded data*, because a block of edges rather than only one edge is loaded at a time. While the edges in a block are independent, they constitute a sub-graph in which information could be propagated by processing it multiple times. In another word, the system could squeeze more value of the loaded data block. This approach is a mid-point between fully synchronous and asynchronous processing and achieves the best of both: ensuring sequential disk I/O by synchronously processing between blocks; and, at the same time, enabling asynchronous processing within each block.

The idea is illustrated in the example in Figure 1 (c). Here, we partition the edges into blocks that each contains two edges, and we apply two computation passes to every loaded block. As a result, the number of iterations is reduced to 2. In the extreme case, if the user further enlarges the loaded data block to contain 6 edges, then only *one* iteration is needed. We call the proposed simple optimization technique *loaded data reentry*. As we see from the SSSP example in Figure 1, loaded data reentry could effectively reduce the number of iterations, reduce the amount of disk I/O and eventually reduce the whole execution time. For each loaded data block, more CPU computation is required. Considering the relative speed of CPU and disk I/O, trading CPU computation for less disk I/O is certainly a sensible choice.

3.2 Beyond the Neighborhood

"Loaded data reentry" is simple and requires only moderate modifications to be applied to existing systems (e.g., GridGraph). However, to apply the principle of "squeezing all the values of loaded data" to more applications, we found that the **neighborhood constraint** imposed by existing systems prohibits the possibility of optimizing in many cases. This neighborhood constraint is enforced by almost all single-machine graph processing systems because in this way one can easily infer the region of data that will be modified by the inputs, which is necessary for disk I/O optimizations. Despite the rationale behind, neighborhood constraint limits the expressiveness of programming model in a way that certain algorithms cannot be implemented in the most efficient manner.

We use weakly connected component (WCC) to explain the problem. WCC is a popular graph problem that calculates whether two arbitrary vertices in a graph are *weakly connected* (i.e., connected after replacing all the directed edges with undirected edges). With the existing programming models, this problem can only be solved by a label-propagation-based algorithm, in which each node repeatedly propagates its current label to its neighbors and update itself if it receives a lower label. The intrinsic property of this algorithm (i.e., the label informa-

tion only propagates one hop in each iteration) inevitably causes the large number of required iterations to coverage, especially for graphs with large diameters. However, if the user-defined function is allowed to update the property of an arbitrary vertex, a disjoint-set [11, 29, 30] data structure can be built in memory. Based on the disjoint-set, WCC problem for any graph can be solved with only *one* pass of the edges.

In general, this method is used in a class of graph algorithms termed *Graph Stream Algorithms* [21], where a graph $G = (V,E)$ is represented as a stream of edges, the storage space of an algorithm is bounded by $O(|V|)$. Graph Stream Algorithms has been studied by the theoretical community for about twenty years [21, 23], and it has been shown that if a randomly accessible $O(|V|)$ space is given, many important graph algorithms can be solved by reading only one (or a few) pass(es) of the graph stream [8]. Unfortunately, the whole class of Graph Stream Algorithms cannot be implemented by the programming model of current disk-based out-of-core systems (or only in a very inefficient manner).

3.3 Limitations

Although the "beyond-neighborhood" algorithms offer significant performance improvements, it also becomes more difficult to infer the range of vertices that will be accessed by a user-defined function. As a result, it becomes more challenging to: *1)* selectively load vertex properties; and *2)* automatically parallelize the execution.

To address the first problem, our solution is to simply mmap all the vertices into memory. While counterintuitive, this straightforward method actually works quite well on many real-world scenarios. In our experiments, we test various data size (up to 6.6B edges) and memory limits (down to only 16MB for small graphs). Results show that our system largely outperforms existing ones in many real-world cases.

The reason of this phenomenon is two-fold. First, the size of vertices is usually considerably smaller than the size of edges but used much more frequently. Our method is deemed to be a good heuristic in memory allocation that naturally reserves as much memory for vertices as possible. Since the density of real-world dataset is usually larger than 30, in typical cases, our method could in fact keep *all* the vertices in memory. This behavior is even valid for industrial-grade workloads. Researchers in Facebook declared in their paper "One Trillion Edges: Graph Processing at Facebook Scale" [7] that industry graphs "can be two orders of magnitude larger" than popular benchmark graphs, which means "hundreds of billions or up to one trillion edges". But, even for such huge graphs, the number of vertices is only about one billion (288M vertices and 60B edges for Twitter, 1.39B vertices and 400B edges for Facebook). This number means that most of the vertices can be cached in memory as the edges typically only need to be read in a stream fashion. This assumption is still valid after using reentry, because we only reentry the loaded edges.

Even more, as discussed in Lin et al. [17], the caching mechanism of mmap is particularly desirable for processing real-world graphs, which often exhibit power-law degree distributions. Our experiment results validate this assumption. Since these high-degree vertices are always cached in memory, accesses to their properties are cheap. In contrast, the other low-degree vertices may be swapped out if the memory limit is low, but they are accessed very infrequently.

As for the second problem, our observation is that: since the complexity of computation is quite low, disk I/O is the *real* bottleneck. It is also confirmed by our results in Section 5: the performance of our single-thread implementation can in fact match the multi-threaded **all-in-memory** systems and is significantly faster than prior multi-threaded *out-of-core* systems. The same phenomenon is also observed by many existing investigations [26], which conclude that there is no need of using multi-threading in an out-of-core environment. To be more general, we also provide a multi-threaded mode in CLIP, which requires users to use atomic operation if necessary. Based on our experience, the increased programming burden is quite limited (only requires the straightforward replacement of the original instruction by the atomic counterpart).

4 CLIP

To support the loaded data reentry and beyond-neighborhood optimization, we design and implement a C++-based novel out-of-core graph processing system, CLIP. CLIP allows users to flexibly write more efficient algorithms that require less number of iterations (and less disk I/O) than algorithms based on previous programming models. The flexibility of our system is achieved due to *1)* its unique execution workflow; and *2)* the ability to break neighborhood constraint. The kernel programming API of CLIP is still "edge function", which is very similar to X-Stream and GridGraph and hence will not much affect the programmability.

4.1 Workflow

CLIP uses the same data model as X-Stream and GridGraph, where the data is modeled as a directed data graph and only the property of vertices can be modified. Figure 2 illuminates the main workflow of CLIP in detail. As for the computation, its procedure is split into two phases. The first phase **sorting** is a pre-processing procedure that sorts all the edges according to a specific order defined by users. We provide a simple interface to al-

low the assignment of the user-defined identifier for each edge. The system will sort edges according to the identifiers. This procedure is typically used to sort edges in grid order[1], where the length of grid is the page size. With this order, the accesses to the property of vertices show good locality. If this standard pre-processing is used, it is the **same** as GridGraph. But, by exposing this API to users, we provide more flexibility. In our experiments, we observe that other orders (e.g., sorting by source only) may be helpful in certain cases (e.g., memory size is enough for caching all the vertices).

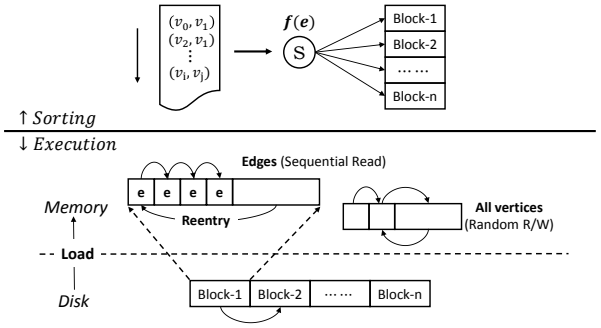


Figure 2: Main workflow of CLIP.

The second phase **execution** is an iterative procedure that circularly reads edges until the property of vertices are converged. Within each iteration, CLIP loads and processes each of the data block by executing the user-defined "edge function" on every edge. Traditional graph processing systems restrict that each data block is processed with only one execution pass in an iteration. In CLIP, each loaded data block is processed by multiple execution passes until all the vertices/edges become inactive. Moreover, we allow users to specify a maximum reentry times (MRT), which is the maximum number of passes that will be executed for every loaded data block. MRT is useful when most further local updating will be invalided by global updating.

4.2 APIs

The programming interface of CLIP is defined in Table 1. This simple API is similar to those provided by existing edge-centric out-of-core systems [26, 41]. **Sort()** and **Exec()** are used to execute **one** iteration of the sorting and execution phase, respectively. To facilitate the users, we also provide a **VMap()** function that iterates every vertex and applies the user-defined input function. Table 1 also defines the type of input parameters and return value of each API function. The input parameter of user-defined function $\mathcal{F}_e$ and $\mathcal{F}_v$ both contain *v_list* with type *Vertices*. *Vertices* is a container by which we can access the property of an arbitrary vertex (mmap-ed into the address space).

Specifically, the input of `Sort()` is a user-defined function $\mathcal{F}_s$ that accepts an edge as input and returns a *double* as the edge's identifier. After the sorting phase, users of CLIP may repeatedly call the function `Exec()` to perform the execution phase for updating the property of vertices. During an iteration, the user-defined function $\mathcal{F}_e$ is applied to edges (potentially multiple times) and can update the property of arbitrary vertices.

Table 1: Programming model of CLIP.

Sort($\mathcal{F}_s$)	—	$\mathcal{F}_s$:= **double** function(*Edge* &*e*)
Exec($\mathcal{F}_e$)	—	$\mathcal{F}_e$:= **void** function(*Vertices* &*v_list*, *Edge* &*e*)
VMap($\mathcal{F}_v$)	—	$\mathcal{F}_v$:= **void** function(*Vertices* &*v_list*, *VertexID* &*vid*)

Our system also supports selective scheduling, which enables us to skip an edge or even a whole block if it is not needed. Specifically, through the *v_list* argument, $\mathcal{F}_e$ can both modify the property of an arbitrary vertex and set its activity. We define that *1)* an edge is inactive if its source vertex is inactive; and *2)* an entire block is inactive if all the edges it contains are inactive. CLIP automatically maintains the activity of every edge/block and uses this information to avoid the unnecessary execution.

4.3 Disk I/O

Although the bandwidth of disk is constantly improving, it still remains as the main bottleneck of out-of-core graph processing systems. Thus, in CLIP, we implement an overlapping mechanism by using a separate loading thread that continuously reads data into a circular buffer until it is full. Moreover, CLIP also enables selective scheduling to further improve the performance. This mechanism is implemented by maintaining the current activity of vertices with a bit-array. With this data structure, CLIP implements two kinds of skipping, namely edge skipping and block skipping. As we have mentioned in Section 4.2, for block skipping, an entire on-disk edge grid will be ignored when it does not contain any active edges (very easy to check bit-array since these source vertices are a continuous range). Moreover, in order to further enable edge skipping, one needs to use `Sort()` function to sort the input edges according to their source vertex. In that case, edges that have the same source vertex will be placed continuously and hence can be skipped at once if this source vertex is inactive (no need of checking the source ID for every edge).

4.4 Examples

To illustrate the usages of CLIP's API, this section presents the implementation of SSSP and WCC,

[1]Grid order means that the adjacent matrix of this graph is cut into grids and the edges belonging to the same grid are stored contiguously. Specifically, an edge (src, dst) is sorted by (src/grid_length, dst/grid_length, src, dst).

which benefit from loaded data reentry and beyond-neighborhood optimization, respectively.

SSSP In SSSP, a property "distance" is attached to each edge and the shortest path is defined as the lowest aggregating distance of all the edges along the path. Similar to other systems, we use a relaxing-based algorithm to solve this problem [5, 9]. Algorithm 1 illustrates the pseudo-code of this algorithm. The `VMap` function is called in the beginning for initialization, which is followed by a series of execution iterations. Each of these iterations executes the same edge function $\mathcal{F}_e$ on every edge, which modifies the distance property of the edge's destination vertex and sets it to active.

Algorithm 1 SSSP Algorithm in CLIP.

```
Functions:
  F_v(v_list, vid) :— {
    if vid == start do
      v_list[vid].dist ← 0;
      v_list.setActive(vid, true);
    else v_list[vid].dist ← INF;
      v_list.setActive(vid, false); }
  F_e(v_list, e) :— {
    dist ← v_list[e.src].dist + e.weight
    if v_list[e.dst].dist > dist do
      v_list[e.dst].dist ← dist;
      v_list.setActive(e.dst, true);
    else v_list.setActive(e.dst, false); }
Computation:
  VMap(F_v);
  Until convergence:
    Exec(F_e);
```

Note that this SSSP implementation is almost the same as original ones, because the trade-off between execution time and disk time is modulated only by MRT. As we will show in Section 5.2.3, the value of MRT is important for achieving a good performance, but it is rather simple to choose an MRT that is good enough.

WCC Different from the label-propagation based algorithm used by prior systems, our algorithm builds a disjoint-set over the property of vertices and uses it to solve WCC for an arbitrary graph with only one iteration. Disjoint-set, also named union-find set, is a data structure that keeps track of a set of elements partitioned into a number of disjoint subsets. It supports two useful operations: *1) find(v)*, which returns an item from *v*'s subset that serves as this subset's representative; and *2) union(u, v)*, which joins the subsets of *u* and *v* into a single subset. Typically, one can check whether two items *u* and *v* belong to the same subset by comparing the results of *find(u)* and *find(v)*. It is guaranteed that if *u* and *v* are from the same subset then *find(u)* == *find(v)*. Otherwise, one can invoke a *union(u, v)* to merge these two subsets.

Algorithm 2 presents the code of our disjoint-set based WCC algorithm. Figure 3 gives an example. In our implementation, each vertex maintains a property *pa* that stores the ID of a vertex. If $pa[u] = v$, we name that the "parent" of vertex *u* is *v*. Vertex *u* is the representative of its subset if and only if $pa[u] = u$. Otherwise, if $pa[u] \neq u$, the representative of *u*'s subset can only be found by going upstream along the *pa* property until finding a vertex that satisfies the above restriction (i.e. function *find* in Algorithm 2). For example, if $pa[3] = 2$, $pa[2] = 1$, $pa[1] = 1$, the subset representative of all these three vertices is 1. The *union* function is implemented by finding the representative of the two input vertices' subset and setting one's *pa* to another. Therefore, the whole procedure of our WCC algorithm can be simply implemented by applying the *union* function to every edge.

Algorithm 2 WCC Algorithm in CLIP.

```
Functions:
  F_find(v_list, vid) :— {
    if v_list[vid].pa == vid do return vid;
    else return v_list[vid].pa =
      F_find(v_list, v_list[vid].pa); }
  F_union(v_list, src, dst) :— {
    s ← F_find(v_list, src);
    d ← F_find(v_list, dst);
    if s < d do v_list[d].pa ← v_list[s].pa;
    else if s > d do v_list[s].pa ← v_list[d].pa; }
  F_e(v_list, e) :— { F_union(v_list, e.src, e.dst); }
  F_v(v_list, vid) :— {
    v_list[vid].pa ← vid;
    v_list.setActive(vid, true); }
Computation:
  VMap(F_v);
  Exec(F_e);
```

In Figure 3 (a), the graph has 4 vertices and 3 edges, the *pa* of every vertex is illustrated by arrows in Figure 3 (b). At the beginning of our algorithm, each vertex belongs to a unique disjoint subset. Hence, all arrows point to their starting vertex (1 in Figure 3(b)). During the execution, the first edge read is $(1,2)$, so their subsets are union-ed by pointing vertex 2's arrow to 1 (2 in Figure 3(b)). In the second step, edge $(2,3)$ is read and their subsets are also union-ed. By going toward upstream of vertex 2's arrow, we can find that its representative is 1. As a result, the union is performed by pointing vertex 3's arrow to vertex 1 (3 in Figure 3(b)). Similarly, the arrow of vertex 4 is redirected to vertex 1 after reading edge $(3,4)$ (4 in Figure 3(b)). Eventually, all arrows point to vertex 1 and hence we found that there is only one weak connected component in the graph.

As one can imagine, this disjoint-set based algorithm always requires only one iteration to calculate WCC for an arbitrary graph, so that it leads to much less work than the original label-propagation based algorithm. But, a potential problem of this algorithm is that, when accessing the property of a vertex, it also needs to access its parent's property (i.e., breaking the neighborhood constraint). Thus, in an extreme case that the property of vertices cannot be all cached and the accesses to parents show great randomness, it may lead to very bad perfor-

mance. However, this problem can be avoided by two simple optimizations: *1)* when calling *union* on two vertices, always uses the vertex that has smaller ID as the parent; and *2)* iterate the edge grids by their x index, which means that the grids are read in the order of "(0, 0), (0, 1), ..., (0, P-1), (1, 0), ..." if the graph edges are partitioned into $P \times P$ grids. According to our evaluation, these two simple optimizations can make sure that most of the parents are stored in the first several pages of vertex property and hence show good locality.

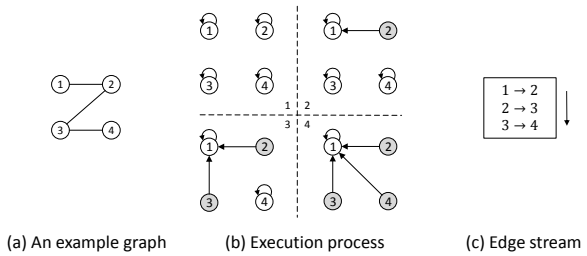


Figure 3: WCC example.

5 Evaluation

In this section, we present our evaluation results on CLIP and compare it with the state-of-art systems X-Stream and GridGraph (as they are reported to be faster than other existing out-of-core graph processing systems like GraphChi). We split all the benchmarks we tested into two categories by their properties and discuss the reason of our speedup respectively.

5.1 Setup

5.1.1 Environment

All our experiments are performed on a single machine that is equipped with two Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz (each has 8-cores), 32GB DRAM (20MB L3 Cache), and a standard 1TB SSD. According to our evaluation, the average throughput of our SSD is about 450MB/s for sequential read. We use a server machine rather than an ordinary PC for the testing because we want to show that the single-thread algorithms implemented in CLIP is even faster than the multi-threaded implementations in X-Stream and GridGraph, which can take advantage of at most 16 threads.

5.1.2 Benchmarks

We consider two categories of benchmarks. The first category is **asynchronous applications**, which includes SSSP, BFS and other algorithms like delta-based PageRank [37], diameter approximation [25], transitive closures [32], betweenness centrality [6], etc. For this kind of applications, the same relaxation based algorithms can be implemented with CLIP as in X-Stream and GridGraph. The only difference is that the user of CLIP can inform the system to enable loaded data reentry by setting MRT. The second category is **beyond-neighborhood applications** (e.g., WCC, MIS), which require users to develop new algorithms to achieve the best performance. One should notice that, for each application, we use either "reentry" **or** "beyond-neighborhood", so that there is no need for a piecewise breakdown of the performance gain.

5.1.3 Methodology

The main performance improvement of CLIP is achieved by reducing the number of iterations with more efficient algorithms. Thus, if all the disk data is cached in memory (which is possible as we have a total of 32GB memory), we cannot observe the impact of disk I/O on overall performance. In order to demonstrate our optimizations in a realistic setting with disk I/O, we use cgroup to set various **memory limits** (from 16MB to 32GB).

Specifically, for every combination of (system, application, dataset), we test three different scenarios: *1)* **all-in-memory**, i.e., limit is set to 32GB so that most of the tested datasets can be fully contained in memory; *2)* **semi-external**, where the memory limit is enough for holding all the vertices but not all the edges; and *3)* **external**, where the memory limit is extremely small so that even vertices cannot be fully held in memory. As the number of vertices and edges are different for different datasets, the thresholds used for semi-external and external are also dataset-specific. The exact numbers are presented in Table 2, from which we can see that the limit is down to only 16MB as the vertex number of LiveJournal is less than 5M.

Table 2: The real-world graph datasets. A random weight is assigned for unweighted graphs.

Graph	Vertices	Edges	Type	Threshold	
				external	semi
LiveJournal [3]	4.85M	69.0M	Directed	16MB	256MB
Dimacs [4]	23.9M	58.3M	Undir.	64MB	256MB
Twitter [14]	41.7M	1.47B	Directed	128MB	4GB
Friendster [2]	65.6M	1.8B	Directed	128MB	4GB
Yahoo [1]	1.4B	6.64B	Directed	4GB	8GB

Moreover, for the clarity of presentation, if not specified explicitly, we always attempt all the possible number of threads and report the best performance. This means that we use at most **16** threads for testing X-Stream and GridGraph. In contrast, we testing CLIP with **16** threads for asynchronous applications but only **one** thread for beyond-neighborhood algorithms.

5.2 Loaded Data Reentry

We use two applications, SSSP and BFS, to evaluate the effect of loaded data reentry technique. All of them can be solved by relaxation based algorithms.

Table 3: Execution time (in seconds) On SSSP/BFS. For each case, we report the results of all three scenarios in the format of "external / semi-external / all-in-memory". '-' is used if we cannot achieve all-in-memory even when the limit is set to 32GB. Since X-Stream requires extra memory for shuffling the messages, 32GB is not enough even for smaller datasets like Friendster and Twitter. '∞' means that the application does not finish after running 24 hours.

		LiveJournal	Dimacs	Friendster	Twitter	Yahoo
SSSP	X-Stream	357.9 / 118.4 / 8.45	77212/ 22647/ 853.2	6352 / 3346 / -	4065 / 2255 / -	∞ / ∞ / -
	GridGraph	66.42 / 48.1 / 6.97	14618/ 13480/ 889.9	1086 / 784.6 / 85.31	1639 / 1083 / 83.51	77298/ 17432/ -
	CLIP	30.14 / 11.23 / 5.09	3202 / 1981 / 316.1	176.2 / 55.79 / 55.85	1353 / 600.6 / 91.82	18160/ 6932 / -
BFS	X-Stream	91.50 / 22.94 / 4.06	8934 / 6538 / 114.9	2526 / 1084 / -	1421 / 627.4 / -	∞ / ∞ / -
	GridGraph	13.20 / 15.4 / 2.49	5199 / 5239 / 406.2	499.6 / 493.7 / 61.54	220.5 / 209.6 / 32.16	35572/ 7403 / -
	CLIP	10.01 / 5.46 / 2.53	1768 / 1059 / 96.12	98.87 / 38.55 / 38.72	141.2 / 110.4 / 44.7	10533/ 3297 / -

5.2.1 Comparison

The results are presented in Table 3, in which all the three different scenarios are included. In this table, '-' means that we cannot achieve *all-in-memory* even when the limit is set to 32GB. and '∞' means that the application does not finish after running 24 hours. As we can see, CLIP can achieve a significant speedup (1.8×-14.06×) under the *semi-external* scenario. In contrast, the speedup on *external* scenario is less (only up to 6.16×). This is reasonable because, with a smaller limit, the number of edges that can be held in memory is less, therefore, the diameter of the sub-graph loaded into memory is smaller. As a result, the effect of reentry is also weaker. Moreover, even for all-in-memory settings, CLIP still outperforms the others if the diameter of the graph is large (e.g., we achieve a 2.7× speedup on Dimacs), which is because that CLIP allows the information to be propagated faster within a sub-graph and eventually makes the convergence faster.

In order to justify the above argument, we compare the number of iterations that is needed for converge on CLIP and the other systems. Results show that our loaded data reentry technique can greatly reduce this number. This improvement is especially significant for large-diameter graphs, like Dimacs, where more than 90% of the iterations can be reduced.

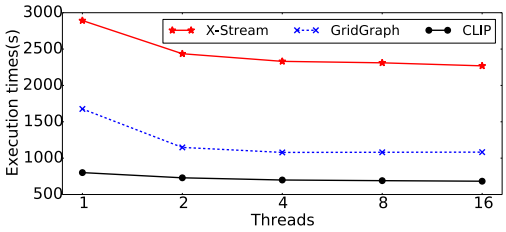


Figure 4: The scalability for SSSP on Twitter graph, evaluated in semi-external scenario.

5.2.2 Scalability

Since we use the same algorithm as X-Stream and GridGraph, our implementation of SSSP and BFS follow the neighborhood constraint. Following neighborhood constraint makes it easy to enable the multi-thread model of CLIP to leverage the multi-core architecture. However, since disk I/O is the real bottleneck, there is actually not a big difference between using multi-thread or not.

Figure 4 illustrates our experiments results on scalability. As we can see, GridGraph has the best scalability as it can achieve a 1.55x speedup by using 4 threads. However, it is large because the single-thread baseline of GridGraph is inefficient. In fact, the single-thread CLIP is already faster than multi-thread version of GridGraph.

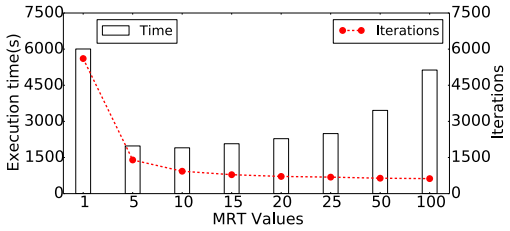


Figure 5: Execution time and required iterations for executing SSSP on Dimacs graph in the semi-external scenario, with different MRT values.

5.2.3 MRT

The value of Maximum Reentry Times (MRT) modulates the trade-off between global updating and local updating. As its effect depends not only on the property of input graph but also on the type of application, there isn't a rule for calculating the best MRT. But, according to our experiences, heuristically setting MRT to 5-10 is usually enough for producing a performance that is matchable with the best possible result (less than 4% difference). For example, all the values we reported in Table 3 is measured at "MRT = 5". The intuitive reason for this phenomena is that the diameter of a real-world graph is typically not large. Figure 5 shows the execution time and required iterations of SSSP on Dimacs graph with different MRTs. We see that both an excessively small MRT (e.g., =1) or an excessively large MRT (≥20) are not helpful. When MRT is larger than 10, while the number of iterations is decreasing, the execution time will actually increase. The reason is that large MRT will lead to many useless iterations within each block, which increases the amount of calculation of CPU without propagating the updates to other blocks.

Table 4: Execution time (in seconds) on WCC and MIS. Format of this table is the same as Table 3. As the size of vertex property is only 1/4 of other applications in MIS, its corresponding thresholds for external and semi-external execution is also only 1/4 of the given number in Table 2, e.g., only 4MB for executing MIS with LiveJournal in external scenario.

		LiveJournal	Dimacs	Friendster	Twitter	Yahoo
WCC	X-Stream	179.5 / 57.77 / 10.25	16633/ 6751 / 185.3	4521 / 2341 / -	1904 / 1194 / -	∞ / ∞ / -
	GridGraph	22.32 / 13.8 / 3.57	6547 / 5757 / 422.5	967.5 / 466.6 / 82.95	431.5 / 272.3 / 62.3	19445/ 2916 / -
	CLIP	3.73 / 2.40 / 2.43	2.61 / 1.35 / 1.33	186 / 65.48 / 64.56	132.7 / 49.03 / 48.85	310.6 / 220.9 / -
MIS	X-Stream	422.1 / 152.6 / 13.06	103.4 / 41.42 / 5.95	9880 / 4867 / -	5513 / 3042 / -	∞ / ∞ / -
	GridGraph	166.6 / 122.1 / 2.98	46.32 / 39.19 / 14.46	3945 / 3777 / 253.7	2510 / 2473 / 156.1	∞ / ∞ / -
	CLIP	6.7 / 2.57 / 2.58	1.6 / 1.17 / 1.21	188.8 / 62.49 / 62.18	90.44 / 49.08 / 49.13	321.5 / 220.2 / -

5.3 Beyond-neighborhood

5.3.1 Applications

For some problems, new algorithms need to be implemented to leverage beyond-neighborhood strategy. Besides WCC that described in Section 4.2, we introduce one more example named MIS in our evaluation.

MIS is an application that finds an **arbitrary** maximal independent set for a graph. In graph theory, a set of vertices constitutes an independent set if and only if any two of these vertices do not have an edge in between. We define that a maximal independent set as a set of vertices that *1)* constitutes an independent set; and *2)* is not a proper subset of any other independent sets. Note that there may be multiple maximal independent sets in a graphs, and MIS only requires to find one arbitrary maximal independent set from them. To solve this problem, X-Stream and GridGraph implement the same parallel algorithm that is based on Monte Carlo algorithm [19]. In contrast, we use a simple greedy algorithm to solve this problem, which consists of three steps: *1)* a *Sort()* is invoked to sort all the edges by their source IDs; *2)* a *VMap()* is called to set the property of all the vertices to *true*; and *3)* an *Exec()* is executed which iterates all the edges in order and set the property *in_mis* of the input edge *e*'s source vertex to *false* if and only if "$e.dst < e.src$ && $v_list[e.dst].in_mis == true$". After executing only **one** time of the *Exec()*, the final results can be obtained by extracting all the vertices whose property *in_mis* are *true*.

Our MIS algorithm is not only beyond-neighborhood but also requires that the edges are processed in a specific order. Thus, it is essentially a sequential algorithm that requires users to use the `Sort()` function provided by CLIP to define a specify pre-processing procedure. However, our algorithm is much faster than the parallel algorithm used by X-Stream and GridGraph, because it requires only one iteration for arbitrary graphs.

5.3.2 Comparison

Table 4 shows the evaluation results on beyond neighborhood applications. We see that CLIP can achieve a significant speed up over the existing systems on all the three scenarios: up to 2508× on *external*, up to 4264× on *semi-external*, and up to 139× on *all-in-memory*. Same as the asynchronous algorithms, the main reason of the speedup in CLIP is that the algorithms require much less iterations to calculate the results. The original algorithms can only converge after using tens or even thousands of iterations. In contrast, out algorithms require only one iteration for all the graphs. As a result, even if we can only use a single thread to execute our beyond-neighborhood algorithms, the large amount of disk I/O and computation avoided by this iteration reduction is enough to offer better performance than other parallel algorithms.

Moreover, as we can see from the table, even though that the algorithms used by CLIP do not follow the neighborhood constraint, they are still much faster than the other systems in the *external* scenario, where the vertices are *not* fully cached in memory. As we have explained in Section 3.3, this is because that the caching mechanism of mmap is particularly suitable for processing power-law graphs. Hence, the number of pages swapping needed for vertices are moderate, at least far less from offsetting the benefit we gain from reducing redundant read of edges.

6 Discussion

6.1 Scope of Application

Although our "reentry" technique is quite simple, it essentially provides a **midpoint** between the fully synchronous algorithm and the fully asynchronous algorithm. It makes the convergence faster than fully synchronous execution but makes an implementation more "disk-friendly" than fully asynchronous execution (i.e. process once a block rather than once a vertex). As a result, all applications that can benefit from asynchronous execution can benefit from "reentry", because they are based on the same principle.

In contrast, the application of "beyond-neighborhood" does rely on the existence of such algorithms. But, according to our study, there are indeed a large set of applications can be optimized with our model. For example, finding WCC of a graph lies at the core of many data mining algorithms, and is a fundamental subroutine in graph clustering. Thus, our method can benefit not only WCC itself but also all these applications. Simi-

larly, MIS shares a similar access pattern of many graph matching applications. In fact, the number of these so-called Graph Stream Algorithms is large enough for publishing a survey on them [8, 21].

Essentially, our "beyond-neighborhood" optimization fundamentally enhances the expressiveness of the vertex programs so that important graph operations like "pointer-jumping" could be implemented. A recent article [16] made the same observation but only discussed it in the context of Galois [24]. This paper shows that such more expressive programming model is not only applicable for in-memory but also feasible for out-of-core graph processing systems. Even more, we argue that the significant performance improvements that "beyond-neighborhood" can achieve also overshadows its limitation on applicability.

6.2 Programmability

McSherry et al. [22] have observed that the scalability of many distributed graph processing system is based on their inefficient single-thread implementation. As a result, they argue that specialized optimized implementations should be used in many real-world scenarios, which share the same principle as our system. However, different from their work that uses a set of distinct programs, CLIP is a complete system that provides a general enough programming model.

The trade-off between more flexibility (potentially worse programmability) and better performance is well-known. Neighborhood-constraint systems choose one extreme of this spectrum, which provides the best programmability but worse performance. McSherry et al.'s work [22] and some others (e.g., Galois [24], smart algorithm in GoFFish [27], Polymer [35]) choose the other extreme. They provide only some basic functionalities (e.g., concurrent loop) or even barely anything. These methods can achieve the best performance, but impose a much larger burden on programmers.

In contrast, we believe that CLIP is a sweet spot in the design space that is just right for out-of-core systems. The slight sacrifice of programmability is definitely worthwhile because this makes CLIP up to tens and even thousands of times faster than existing systems. According to our evaluation, the programming model of CLIP helps us to write all the programs described in this paper in less than 80 lines of codes, comparing to 1200 lines for the native algorithms (many lines of code are used for dealing with chores like I/O, partitioning, etc.).

6.3 Compared with In-memory System

Thanks to flexibility of CLIP, its performance on many kinds of applications is matchable with in-memory systems. As an illustration, Table 5 presents the comparison between CLIP (semi-external mode) and manually optimized algorithms that implemented in Galois. Since the loading of data dominates the execution time, the performance of CLIP is indeed comparable to Galois. CLIP is slower than Galois on large datasets (Friendster, Twitter) because we use different encoding formats for the binary graph file on disk. Take "Twitter" as an example, the input edges size of WCC is 11.25GB for Galois but 21.88GB for CLIP.

Besides Galois, GraphMat [28] is also an in-memory graph processing system that takes advantage from efficient matrix operations. According to our evaluation, GraphMat requires only 0.72s to calculate the WCC of LiveJournal, which is faster than both Galois and CLIP (while it requires 9.78s for loading data). However, GraphMat employs a synchronous execution engine that enforces neighborhood constraint. Thus, for graphs that have a large diameter, its performance is poor. For example, GraphMat needs 6262 iterations (221.9s) to achieve the convergence of WCC algorithm on Dimacs (only 1 iteration and 1.35s are needed for CLIP).

Table 5: Execution time (in seconds) for CLIP and Galois. '-' designates out of memory.

	LiveJournal	Dimacs	Friendster	Twitter	Yahoo
WCC					
Galois	2.58	1.81	49.75	42.36	-
CLIP	2.4	1.35	65.48	49.03	220.9
MIS					
Galois	2.01	1.36	40.14	34.15	-
CLIP	2.57	1.17	62.49	49.08	220.2

6.4 Concurrency

As mentioned in Section 5.2.2, users of CLIP can enable multi-thread execution for applications that voluntarily obey the neighborhood constraint (e.g., SSSP). Specifically, for executing `VMap()` in parallel, the whole vertex set is split into equal intervals that are dispatched to different worker threads. Similarly, for executing `Exec()`, the loaded edge grid is further split and dispatched. With neighborhood constraint, the concurrency control can be implemented by fined-grained locking in a straightforward manner. However, although the locking mechanism can assure the correctness of our system, certain downsides of asynchronous execution still exist in CLIP, such as non-deterministic execution and unstable performance. However, asynchronous execution has been demonstrated to be able to accelerate the convergence of iterative computations [10].

Besides multi-threads, there are also some graph systems that support multi-tenant execution [7, 20]. Different from them, CLIP is a single machine graph processing system and does not support multi-tenant execution, which is similar to prior systems [15, 26, 41]. Typically, multi-tenant is more useful for distributed systems that share the same cluster.

6.5 Evaluation on HDD

It is worth mentioning that, CLIP can also achieve a good performance on slow storage devices (e.g., HDD). We evaluate CLIP on a standard 3TB HDD and compare it with X-Stream and GridGraph. According to our evaluation, the average throughput of our HDD is about 150MB/s for sequential read. Table 6 shows the evaluation results under the semi-external scenario. Since the amount of loading data dominates the execution time, CLIP can achieve a similar or even better speedup (5.59×-5999× for WCC, 2.32×-15.37× for BFS) with the evaluation on SSD.

Table 6: Execution time (in seconds) on HDD. '∞' means that the application does not finish after running 24 hours.

	LiveJournal	Dimacs	Friendster	Twitter	Yahoo
WCC					
X-Stream	128.1	15417	5219	2519	∞
GridGraph	34.68	16467	1314	785.9	8764
CLIP	6.2	2.57	160.2	132.1	590.4
BFS					
X-Stream	53.25	16943	2566	1067	∞
GridGraph	34.28	12790	1431	604.8	22528
CLIP	14.77	2659	93.1	217.3	8844

6.6 Preprocessing Time

Pre-processing is a necessary procedure for most (e.g., GraphChi, GridGraph, CLIP) but not all (e.g., X-Stream) out-of-core graph processing systems. The pre-processing cost of CLIP is similar to GridGraph, as they are almost the same. Moreover, although sometimes the pre-processing time is longer than the execution time, it is still worthwhile in terms of total execution time. For example, the total execution time (pre-processing+computation) of computing MIS on Friendster is 4867s for X-Stream and 3962.5s for GridGraph. In contrast, the total execution time of CLIP is 145.3s for pre-processing and only 62.49s for computation, which in total is 207.79s. As we can see, the total execution time of CLIP is 19.07× faster than GridGraph and 23.42× faster than X-Stream, not to mention that the pre-processing cost can be amortized by reusing the results.

7 Related Work

There are also many distributed graph processing systems. Pregel [20] is the earliest distributed graph processing system that proposes a vertex-centric programming model, which is later inherited by many other graph processing systems [12, 18, 26, 36, 40]. Some existing works [27, 31], such as Giraph++ [32], have suggested to replace "think as vertex" with "think as subgrapg/partition/embedding". They can take advantage of the fact that each machine contains a subset of data rather than only one vertex/edge and hence are much faster than prior works. However, none of these existing works could support the beyond-neighborhood algorithms used by CLIP.

Similarly, in addition to GraphChi, X-Stream and GridGraph, there are other out-of-core graph processing systems using alternative approaches [13, 17, 38, 39]. However, most of them only focus on maximizing the locality of disk I/O and still use neighborhood-constraint programming model. As a counter example, MMap [17] leverages the memory mapping capability found on operating systems by mapping edge and vertex data files in memory, which inspires the design of CLIP. But, MMap only demonstrates that mmap's caching mechanism is naturally suitable for processing power-law graphs. It does not consider the limitations of the original out-of-core systems' restrictions , which is the key contribution of this work.

There are some works [34, 39] that aim to load only necessary data in an iteration, which can also reduce disk I/O. However, these methods are actually an orthogonal optimization with our efforts of reducing the number of iterations. According to our evaluation, our simple selective scheduling method is enough for our case.

Some existing works [15, 33] are proposed to support evolving graphs, which is not currently supported in our system. But, although it is not discussed, the same mechanism for dealing with evolving graph in GraphChi can be added to CLIP in a straightforward manner. To maintain the consistency of data, we reserve all the addition and deletion of edges within an iteration and only apply them in the interval between two iterations.

8 Conclusion

In this paper, we propose CLIP, a novel out-of-core graph processing system designed with the principle of "squeezing out all the value of loaded data". With the more expressive programming model and more flexible execution, CLIP enables more efficient algorithms that require much less amount of total disk I/O. Our experiment results show that CLIP is up to tens or sometimes even thousands times faster than existing works X-Stream and GridGraph.

Acknowledgement This work is supported by National Key Research & Development Program of China (2016YFB1000504), Natural Science Foundation of China (61433008, 61373145, 61572280, 61133004, 61502019, U1435216), National Basic Research (973) Program of China (2014CB340402), Intel Labs China (Funding No.20160520). This work is also supported by NSF CRII-1657333, NSF SHF-1717754, NSF CSR-1717984, Spanish Gov. & European ERDF under TIN2010-21291-C02-01 and Consolider CSD2007-00050. Contact: Yongwei Wu (wuyw@tsinghua.edu.cn) and Kang Chen (chenkang@tsinghua.edu.cn).

References

[1] G2 - Yahoo! AltaVista Web Page Hyperlink Connectivity Graph, circa 2002. http://webscope. sandbox.yahoo.com/.

[2] S. N. A. Project. Stanford large network dataset collection. http://snap.stanford.edu/data/com-Friendster.html.

[3] S. N. A. Project. Stanford large network dataset collection. http://snap.stanford.edu/data/soc-LiveJournal1.html.

[4] The Center for Discrete Mathematics and Theoretical Computer Science. http://www.dis.uniroma1.it /challenge9/download.shtml.

[5] Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, pages 87–90, 1958.

[6] Ulrik Brandes. A faster algorithm for betweenness centrality*. *Journal of mathematical sociology*, 25(2):163–177, 2001.

[7] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: graph processing at Facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[8] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. In *International Colloquium on Automata, Languages, and Programming*, pages 531–543. Springer, 2004.

[9] Michael L Fredman. New bounds on the complexity of the shortest path problem. *SIAM Journal on Computing*, 5(1):83–89, 1976.

[10] Andreas Frommer and Daniel B Szyld. On Asynchronous Iterations. 1999.

[11] Harold N Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of computer and system sciences*, 30(2):209–221, 1985.

[12] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 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)*, pages 17–30, 2012.

[13] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 77–85. ACM, 2013.

[14] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600. ACM, 2010.

[15] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: large-scale graph computation on just a PC. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, 2012.

[16] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Parallel Graph Analytics. *Communication of ACM*, 59(5):78–87, 2016.

[17] Zhiyuan Lin, Minsuk Kahng, Kaeser Md Sabrin, Duen Horng Polo Chau, Ho Lee, and U Kang. Mmap: Fast billion-scale graph computation on a pc via memory mapping. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 159–164. IEEE, 2014.

[18] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[19] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.

[20] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[21] Andrew McGregor. Graph stream algorithms: a survey. *ACM SIGMOD Record*, 43(1):9–20, 2014.

[22] Frank McSherry, Michael Isard, and Derek G Murray. Scalability! But at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[23] Shanmugavelayutham Muthukrishnan. *Data streams: Algorithms and applications*. Now Publishers Inc, 2005.

[24] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.

[25] Liam Roditty and Virginia Vassilevska Williams. Fast approximation algorithms for the diameter and radius of sparse graphs. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 515–524. ACM, 2013.

[26] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[27] Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. Goffish: A sub-graph centric framework for large-scale graph analytics. In *European Conference on Parallel Processing*, pages 451–462. Springer, 2014.

[28] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. GraphMat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment*, 8(11):1214–1225, 2015.

[29] Robert E Tarjan and Jan Van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM (JACM)*, 31(2):245–281, 1984.

[30] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.

[31] Carlos HC Teixeira, Alexandre J Fonseca, Marco Serafini, Georgos Siganos, Mohammed J Zaki, and Ashraf Aboulnaga. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 425–440. ACM, 2015.

[32] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.

[33] Keval Vora, Rajiv Gupta, and Guoqing Xu. Synergistic Analysis of Evolving Graphs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(4):32, 2016.

[34] Keval Vora, Guoqing Xu, and Rajiv Gupta. Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, 2016.

[35] Kaiyuan Zhang, Rong Chen, and Haibo Chen. NUMA-aware graph-structured analytics. In *ACM SIGPLAN Notices*, volume 50, pages 183–193. ACM, 2015.

[36] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. Exploring the hidden dimension in graph processing. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 285–300. USENIX Association, 2016.

[37] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Accelerate large-scale iterative computation through asynchronous accumulative updates. In *Proceedings of the 3rd workshop on Scientific Cloud Computing Date*, pages 13–22. ACM, 2012.

[38] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, and Matei Zaharia Saman Amarasinghe. Optimizing Cache Performance for Graph Analytics. *arXiv preprint arXiv:1608.01362*, 2016.

[39] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, 2015.

[40] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)(Savannah, GA*, 2016.

[41] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 375–386, 2015.

Ending the Anomaly: Achieving Low Latency and Airtime Fairness in WiFi

Toke Høiland-Jørgensen
Karlstad University

Michał Kazior
Tieto Poland

Dave Täht
TekLibre

Per Hurtig
Karlstad University

Anna Brunstrom
Karlstad University

Abstract

With more devices connected, delays and jitter at the WiFi hop become more prevalent, and correct functioning during network congestion becomes more important. However, two important performance issues prevent modern WiFi from reaching its potential: increased latency under load caused by excessive queueing (i.e. *bufferbloat*) and the 802.11 performance anomaly.

To remedy these issues, we present a novel two-part solution. We design a new queueing scheme that eliminates bufferbloat in the wireless setting. Leveraging this queueing scheme, we then design an airtime fairness scheduler that operates at the access point and doesn't require any changes to clients.

We evaluate our solution using both a theoretical model and experiments in a testbed environment, formulating a suitable analytical model in the process. We show that our solution achieves an order of magnitude reduction in latency under load, large improvements in multi-station throughput, and nearly perfect airtime fairness for both TCP and downstream UDP traffic. Further experiments with application traffic confirm that the solution provides significant performance gains for real-world traffic.We develop a production quality implementation of our solution in the Linux kernel, the platform powering most access points outside of the managed enterprise setting. The implementation has been accepted into the mainline kernel distribution, making it available for deployment on billions of devices running Linux today.

1 Introduction

As more mobile devices connect to the internet, and internet connections increase in capacity, WiFi is increasingly the bottleneck for users of the internet. This means that congestion at the WiFi hop becomes more common, which in turn increases the potential for bufferbloat at the WiFi link, severely degrading performance [10].

The 802.11 performance anomaly [9] also negatively affects the performance of WiFi bottleneck links. This is a well-known property of WiFi networks: if devices on the network operate at different rates, the MAC protocol will ensure *throughput fairness* between them, meaning that all stations will effectively transmit at the lowest rate. The anomaly was first described in 2003, and several mitigation strategies have been proposed in the literature (e.g., [13, 26]), so one would expect the problem to be solved. However, none of the proposed solutions have seen widespread real-world deployment.

Recognising that the solutions to these two problems are complementary, we design a novel queue management scheme that innovates upon previous solutions to the bufferbloat problem by adapting it to support the 802.11 suite of WiFi protocols. With this queueing structure in place, eliminating the performance anomaly becomes possible by scheduling the queues appropriately. We develop a deficit-based airtime fairness scheduler to achieve this.

We implement our solution in the WiFi stack of the Linux kernel. Linux is perhaps the most widespread platform for commercial off-the-shelf routers and access points outside the managed enterprise, and hundreds of millions of users connect to the internet through a Linux-based gateway or access point on a daily basis. Thus, while our solution is generally applicable to any platform that needs to support WiFi, using Linux as our example platform makes it possible to validate that our solution is of production quality, and in addition gives valuable insights into the practical difficulties of implementing these concepts in a real system.

The rest of this paper describes our solution in detail, and is structured as follows: Section 2 describes the bufferbloat problem in the context of WiFi and the WiFi performance anomaly, and shows the potential performance improvement from resolving them. Section 3 describes our proposed solution in detail and Section 4 presents our experimental evaluation. Finally, Section 5 summarises related work and Section 6 concludes.

2 Background

In this section we describe the two performance issues we are trying to solve – Bufferbloat in the WiFi stack and the 802.11 performance anomaly. We explain why these matter, and show the potential benefits from solving them.

2.1 Bufferbloat in the context of WiFi

Previous work on eliminating bufferbloat has shown that the default buffer sizing in many devices causes large delays and degrades performance. It also shows that this can be rectified by introducing modern queue management to the bottleneck link [10, 15, 29]. However, this does not work as well for WiFi; prior work has shown that neither decreasing buffer sizes [23] nor applying queue management algorithms to the WiFi interface [10] can provide the same reduction in latency under load as for wired links.

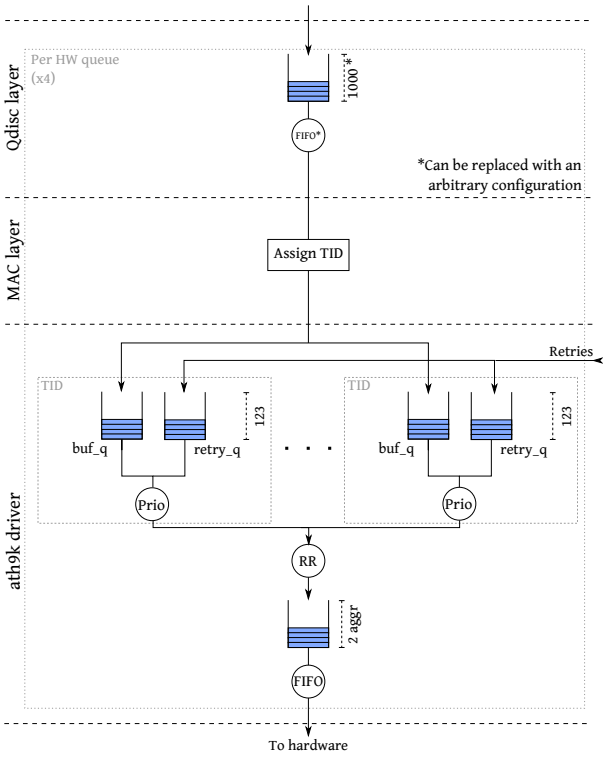


Figure 1: The queueing structure of the Linux WiFi stack.

The reason for the limited effect of prior solutions is queueing in the lower layers of the wireless network stack. For Linux, this is clearly seen in the queueing structure, depicted in Figure 1. The upper queue discipline ("qdisc") layer, which is where the advanced queue management schemes can be installed, sits above both the mac80211 subsystem (which implements the base 802.11 protocol) and the driver. As the diagram shows, there is significant unmanaged queueing in these lower layers, limiting the efficacy of the queue management schemes and leading to increased delay. Such a design is typical for an environment where low-level protocol details impose a certain queueing structure (as opposed to a wired Ethernet network, where the protocol-specific processing performed by the driver does not necessitate queueing). In WiFi this queueing is needed to build aggregates (and to a lesser extent to keep the hardware busy within the time constrains imposed by the protocol), but a similar situation can be seen in, e.g., mobile broadband devices, DSL modem drivers, and even in some VPN protocols, where the encryption processing can require a separate layer of queueing.

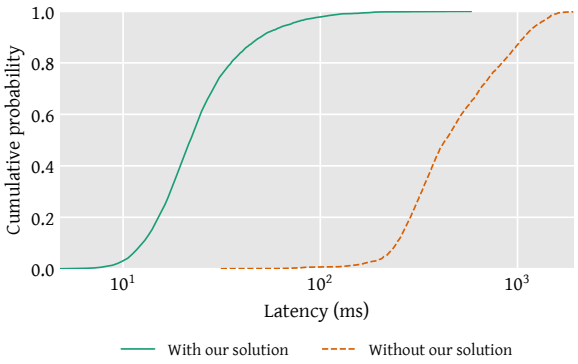


Figure 2: Latency of an ICMP ping flow with simultaneous TCP download traffic, before and after our modifications.

To solve this, an integrated queueing scheme is needed, that applies modern queue management to the protocol-specific queueing structures. In Section 3 we describe our design of such a solution for the WiFi domain. Figure 2 showcases the gain from applying our solution. The figure shows a latency measurement (ICMP ping) performed simultaneously with a simple TCP download to each of the stations on the network. The dashed line shows the state of the Linux kernel before we applied our solution, with several hundred milliseconds of added latency. The solid line shows the effects of applying the solution we propose in this paper – a latency reduction of an order of magnitude.

2.2 Airtime fairness

The 802.11 performance anomaly was first described for the 802.11b standard in [9], which showed that in a wireless network with differing rates, each station would achieve the same effective throughput even when their rates were different. Later work has shown both analytically and experimentally that time-based fairness improves the aggregate performance of the network [26], and

that the traditional notion of proportional fairness [18] translates to airtime fairness when applied to a WiFi network [12].

This latter point is an important part of why airtime fairness is desirable – proportional fairness strikes a balance between network efficiency and allowing all users a minimal level of service. Since a wireless network operates over a shared medium (the airwaves), access to this medium is the scarce resource that needs to be regulated. Achieving airtime fairness also has the desirable property that it makes a station's performance dependent on the *number* of active stations in the network, and not on the performance of each of those other stations.

The addition of packet aggregation to WiFi (introduced in 802.11n and also present in 802.11ac) adds some complexity to the picture. To quantify the expected gains of airtime fairness in the context of these newer revisions of 802.11, the following section develops an analytical model to predict throughput and airtime usage.

2.2.1 An analytical model for 802.11 with aggregation

The models in [9] and [26] give analytical expressions for expected throughput and airtime share for 802.11b (the latter also under the assumption of airtime fairness). Later work [16] updates this by developing analytical expressions for packet sizes and transmission times for a single station using 802.11n. However, this work does not provide expressions for predicting throughput and airtime usage. In this section we expand on the work of [16] to provide such an expression. While we focus on 802.11n here, the 802.11ac standard is backwards-compatible with 802.11n as far as the aggregation format is concerned, so these calculations apply to the newer standard as well.

For the following exposition, we assume a set of active stations, I. Each station, i, transmits aggregates of a fixed size of L_i bytes. In practice, the aggregates are composed of data packets, plus overhead and padding. The 802.11n standard permits two types of aggregation (known as A-MPDU and A-MSDU), which differ in how they combine packets into MAC-layer aggregates. For A-MPDU aggregation (which is the most common in use in 802.11n devices), the size of an aggregate consisting of n_i packets of size l_i is given by:

$$L_i = n_i(l_i + L_{delim} + L_{mac} + L_{FCS} + L_{pad}) \quad (1)$$

where $L_{delim}, L_{mac}, L_{FCS}, L_{pad}$ are, respectively, the frame delimiter, MAC header, frame check sequence and frame padding. However, these details are not strictly necessary for our exposition, so we leave them out in the following and instead refer to [16] for a nice overview of the details of aggregate composition.

A station transmits data over the air at a particular data rate r_i (measured in bits per second). So the time to transmit the data portion of an aggregate is simply:

$$T_{data}(i) = \frac{8L_i}{r_i} \quad (2)$$

From this we can compute the expected effective station rate, assuming no errors or collisions, and no other active stations:

$$R_0(i) = \frac{L_i}{T_{data}(i) + T_{oh}} \quad (3)$$

where T_{oh} is the per-transmission overhead, which consists of the frame header, the inter-frame spacing, the average block acknowledgement time, and the average back-off time before transmission. We again leave out the details and point interested readers to [2, 16].

Turning to airtime fairness, we borrow two insights from the analysis in [26]:

1. The rate achieved by station i is simply given by the baseline rate it can achieve when no other stations are present (i.e., $R_0(i)$) multiplied by the share of airtime available to the station.

2. When airtime fairness is enforced, the airtime is divided equally among the stations (by assumption). When it is not, the airtime share of station i is the ratio between the time that station spends on a single transmission (i.e., $T_{data}(i)$) and the total time all stations spend doing one

Aggr size	$T(i)$	Rates (Mbps)			
		PHY	Base	$R(i)$	Exp
Baseline (FIFO queue)[1]					
6892	10%	144.4	97.3	9.7	7.1
7833	11%	144.4	101.1	11.4	6.3
2914	79%	7.2	6.5	5.1	5.3
Total				26.4	18.7
Airtime Fairness					
28434	33%	144.4	126.7	42.2	38.8
28557	33%	144.4	126.8	42.3	35.6
2914	33%	7.2	6.5	2.2	2.0
Total				86.8	76.4

Table 1: Calculated airtime, calculated rate and measured rate for the three stations (two fast and one slow) in our experimental setup. The aggregation size is the measured mean aggregation size (in bytes) from our experiments and the measured rates (Exp column) are mean UDP throughput values.

[1] The aggregation size and throughput values vary quite a bit for this test, because of the randomness of the FIFO queue emptying and filling. We use the median value over all repetitions of the per-test mean throughput and aggregation size; see the online appendix for graphs with error bars.

transmission each.

With these points in mind, we express the expected airtime share $T(i)$ and rate $R(i)$ as:

$$T(i) = \begin{cases} \frac{1}{|I|} & \text{with fairness} \\ \frac{T_{data}(i)}{\sum_{j \in I} T_{data}(j)} & \text{otherwise} \end{cases} \quad (4)$$

$$R(i) = T(i) R_0(i) \quad (5)$$

Using the above, we can calculate the expected airtime share and effective rate for each station in our experimental setup. The assumption of no contention holds because all data is transmitted from the access point. As the queueing structure affects the achievable aggregation level (and thus the predictions of the model), we use the measured average aggregation levels in our experiments as input to the model.

The model predictions, along with the actual measured throughput in our experiments, are shown in Table 1. The values will be discussed in more detail in Section 4, so for now we will just remark that this clearly shows the potential of eliminating the performance anomaly: An increase in total throughput by up to a factor of five.

3 Our solution

We focus on the access point scenario in formulating our solution, since a solution that only requires modifying the access point makes deployment easier as there are fewer devices to upgrade. However, WiFi client devices can also benefit from the proposed queueing structure. And while we have focused on 802.11n here, the principles apply equally to both earlier (802.11abg) and newer (802.11ac) standards. The rest of this section describes the two parts of our solution, and outlines the current implementation status in Linux.

3.1 A bloat-free queueing structure for 802.11

An operating system networking stack has many layers of intermediate queueing between different subsystems, each of which can add latency. For specialised systems, it is possible to remove those queues entirely, which achieves significant latency reductions [1]. While such a radical restructuring of the operating system is not always possible, the general principle of collapsing multiple layers of queues can be applied to the problem of reducing bufferbloat in WiFi.

As mentioned in Section 2.1, an integrated queueing structure is needed to deal with protocol-specific constraints while still eliminating bufferbloat. What we propose here is such an integrated structure that is specifically suited to the 802.11 MAC. The components we use to build this structure already exists in various forms; the novelty of our solution lies in their integration, and some algorithmic innovations to make the implementation feasible, even on small devices.

There are three main constraints we must take into account when designing our queueing scheme. First, we must be able to handle aggregation; the 802.11e standard specifies that packets can be assigned different Traffic Identifiers (TIDs) (typically based on their DiffServ markings [25]), and the 802.11n standard specifies that aggregation be performed on a per-TID basis. Second, we must have enough data processed and ready to go when the hardware wins a transmit opportunity; there is not enough time to do a lot of processing at that time. Third, we must be able to handle packets that are re-injected from the hardware after a failed transmission; these must be retransmitted ahead of other queued packets, as transmission can otherwise stall due to a full Block Acknowledgement Window.

The need to support aggregation, in particular, has influenced our proposed design. A generic packet queueing mechanism, such as that in the Linux qdisc layer (see Section 2.1), does not have the protocol-specific knowledge to support the splitting of packets into separate queues, as is required for aggregation. And introducing an API to communicate this knowledge to the qdisc layer would impose a large complexity cost on this layer, to the detriment of network interfaces that do not have the protocol-specific requirements. So rather than modifying the generic queueing layer, we bypass it completely, and instead incorporate the smart queue management directly into the 802.11 protocol-specific subsystem. The main drawback of doing this is, of course, a loss of flexibility. With this design, there is no longer a way to turn off the smart queue management completely; and it does add some overhead to the packet processing. However, as we will see in the evaluation section, the benefits by far outweigh the costs.

We build our smart queue management solution on the FQ-CoDel queue management scheme, which has been shown to be a best-in-class bufferbloat mitigation technique [10, 15, 29]. The original FQ-Codel algorithm is a hybrid fairness queueing and AQM algorithm [11]. It functions as a Deficit Round-Robin (DRR) scheduler [24] between flows, hashing packets into queues based on their transport protocol flows, and applying the CoDel AQM separately to each queue, in order to keep the latency experienced by each flow under control. FQ-CoDel also adds an optimisation for sparse flows to the basic DRR algorithm. This optimisation allows flows that use less than their fair share of traffic to gain scheduling priority, reducing the time their packets spend in the queue. For a full explanation of FQ-CoDel, see [11].

FQ-CoDel allocates a number of sub-queues that are used for per-flow scheduling, and so simply assigning a full instance of FQ-CoDel to each TID is impractical. In-

Algorithm 1 802.11 queue management algorithm - enqueue.

```
1:  function enqueue(pkt, tid)
2:      if queue_limit_reached() then                ▷ Global limit
3:          drop_queue ← find_longest_queue()
4:          drop(drop_queue.head_pkt)
5:      queue ← hash(pkt)
6:      if queue.tid ≠ NULL and queue.tid ≠ tid then
7:          queue ← tid.overflow_queue               ▷ Hash collision
8:      queue.tid ← tid
9:      timestamp(pkt)                   ▷ Used by CoDel at dequeue
10:     append(pkt, queue)
11:     if queue is not active then
12:         list_add(queue, tid.new_queues)
```

stead, we innovate on the FQ-CoDel design by having it operate on a fixed total number of queues, and group queues based on which TID they are associated with. So when a packet is hashed and assigned to a queue, that queue is in turn assigned to the TID the packet is destined for. In case that queue is already active and assigned to another TID (which means that a hash collision has occurred), the packet is instead queued to a TID-specific overflow queue.[2] A global queue size limit is kept, and when this is exceeded, packets are dropped from the globally longest queue, which prevents a single flow from locking out other flows on overload. The full enqueue logic is shown in Algorithm 1.

The lists of active queues are kept in a per-TID structure, and when a TID needs to dequeue a packet, the FQ-CoDel scheduler is applied to the TID-specific lists of active queues. This is shown in Algorithm 2.

The obvious way to handle the two other constraints mentioned above (keeping the hardware busy, and handling retries), is, respectively, to add a small queue of preprocessed aggregates, and to add a separate priority queue for packets that need to be retried. And indeed, this is how the ath9k driver already handled these issues, so we simply keep this. The resulting queueing structure is depicted in Figure 3.

3.2 Airtime fairness scheduling

Given the above queueing structure, achieving airtime fairness becomes a matter of measuring the airtime used by each station, and appropriately scheduling the order in which stations are served. For each packet sent or received, the packet duration can either be extracted directly from a hardware register, or it can be calculated from the packet length and the rate at which it was sent (including any retries). Each packet's duration is subtracted

[2] A hash collision can of course also mean that two flows assigned to the same TID end up in the same queue. In this case, no special handling is needed, and the two flows will simply share a queue like in any other hash-based fairness queueing scheme.

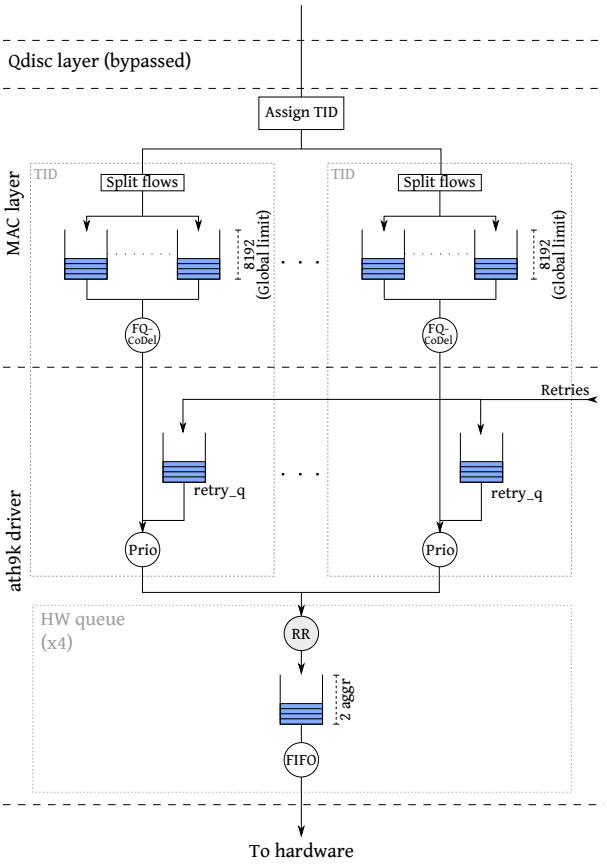


Figure 3: Our 802.11-specific queueing structure, as it looks when applied to the Linux WiFi stack.

from a per-station airtime deficit which is used by a deficit scheduler, modelled after FQ-CoDel, to decide the destination station ahead of each transmission. The decision to keep the deficit per station instead of per TID follows from the fact that the goal of airtime fairness is to even out differences in the physical signal conditions, which is a per-station property. However, because the four 802.11 QoS precedence markings (VO, VI, BE and BK) are commonly scheduled independently down to the hardware level, we actually keep four deficits per station, corresponding to the four precedence levels, to simplify the scheduler implementation.

The resulting airtime fairness scheduler is shown in Algorithm 3. It is similar to the the FQ-CoDel dequeue algorithm, with stations taking the place of flows, and the deficit being accounted in microseconds instead of bytes. The two main differences are (1) that the scheduler function loops until the hardware queue becomes full (at two queued aggregates), rather than just dequeueing a single packet; and (2) that when a station is chosen to be scheduled, it gets to build a full aggregate rather than a single packet.

Compared to the closest previously proposed solu-

Algorithm 2 802.11 queue management algorithm - dequeue.

```
1:  function dequeue(tid)
2:      if tid.new_queues is non-empty then
3:          queue ← list_first(tid.new_queues)
4:      else if tid.old_queues is non-empty then
5:          queue ← list_first(tid.old_queues)
6:      else
7:          return NULL
8:      if queue.deficit ≤ 0 then
9:          queue.deficit ← queue.deficit + quantum
10:         list_move(queue, tid.old_queues)
11:         restart
12:     pkt ← codel_dequeue(queue)
13:     if pkt is NULL then                    ▷ queue empty
14:         if queue ∈ tid.new_queues then
15:             list_move(queue, tid.old_queues)
16:         else
17:             list_del(queue)
18:             queue.tid ← NULL
19:         restart
20:     queue.deficit ← queue.deficit − pkt.length
21:     return pkt
```

Algorithm 3 Airtime fairness scheduler. The schedule function is called on packet arrival and on transmission completion.

```
1:  function schedule
2:      while hardware queue is not full do
3:          if new_stations is non-empty then
4:              station ← list_first(new_stations)
5:          else if old_stations is non-empty then
6:              station ← list_first(old_stations)
7:          else
8:              return
9:          deficit ← station.deficit[pkt.qoslvl]
10:         if deficit ≤ 0 then
11:             station.deficit[pkt.qoslvl] ← deficit + quantum
12:             list_move(station, old_stations)
13:             restart
14:         if station's queue is empty then
15:             if station ∈ new_stations then
16:                 list_move(station, old_stations)
17:             else
18:                 list_del(station)
19:             restart
20:         build_aggregate(station)
```

tion [6], our scheme has several advantages:

1. We lower implementation complexity by leveraging existing information on per-aggregate transmission rates and time, and by using a per-station deficit instead of token buckets, which means that no token bucket accounting needs to be performed at TX and RX completion time.

2. [6] measures time from an aggregate is submitted to the hardware until it is sent, which risks including time spent waiting for other stations to transmit. We increase accuracy by measuring the actual time spent transmitting, and by also accounting the airtime from received frames to each station's deficit.

3. We improve on the basic scheduler design by adding an optimisation for sparse stations, analogous to FQ-CoDel's sparse flow optimisation. This improves latency for stations that only transmit occasionally, by giving them temporary priority for one round of scheduling. We apply the same protection against gaming this mechanism that FQ-CoDel does to its sparse flow mechanism [11].

3.3 Implementation

We have implemented our proposed queueing scheme in the Linux kernel, modifying the mac80211 subsystem to include the queueing structure itself, and modifying the ath9k and ath10k drivers for Qualcomm Atheros 802.11n and 802.11ac chipsets to use the new queueing structure. The airtime fairness scheduler implementation is limited to the ath9k driver, as the ath10k driver lacks the required scheduling hooks.

Our modifications have been accepted into the mainline Linux kernel, different parts going into kernel releases 4.8 through 4.11, and is included in the LEDE open source router firmware from release 17.01. The implementation is available online, as well as details about our test environment and the full evaluation dataset.[3]

4 Evaluation

We evaluate our modifications in a testbed setup consisting of five PCs: Three wireless clients, an access point, and a server located one Gigabit Ethernet hop from the access point, which serves as source and sink for the test flows. All the wireless nodes are regular x86 PCs equipped with PCI-Express Qualcomm Atheros AR9580 adapters (which use the ath9k driver). Two of the test clients are placed in close proximity to the access point (and are referred to as fast nodes), while the last (referred to as the slow node) is placed further away and configured to only support the MCS0 rate, giving a maximum throughput to that station of 7.2 Mbps at the PHY layer. A fourth virtual station is added as an additional fast node to evaluate the sparse station optimisation (see Section 4.1.4 below). All tests are run in HT20 mode on an otherwise unused channel in the 5Ghz band. We use 30 test repetitions of 30 seconds each unless noted otherwise.

The wireless nodes run an unmodified Ubuntu 16.04 distribution. The access point has had its kernel replaced with a version 4.6 kernel from kernel.org on top of which we apply our modifications. We run all experiments with four queue management schemes, as follows:

[3] See `http://www.cs.kau.se/tohojo/airtime-fairness/` for the online appendix that contains additional material, as well as the full experimental dataset and links to the relevant Linux code.

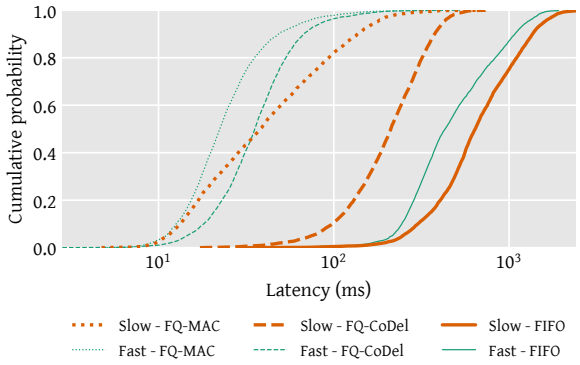


Figure 4: Latency (ICMP ping) with simultaneous TCP download traffic.

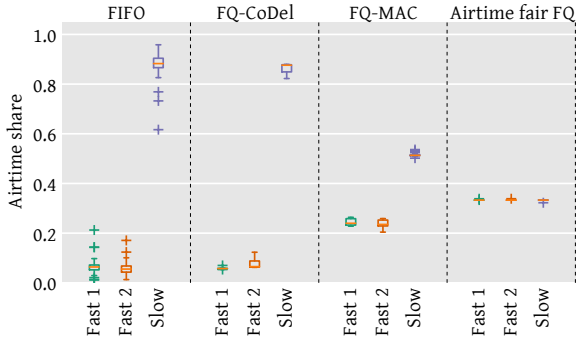


Figure 5: Airtime usage for one-way UDP traffic. Each column shows the relative airtime usage of one of the three stations, with the four sections corresponding to the four queue management schemes.

FIFO: The default 4.6 kernel from kernel.org modified only to collect the airtime used by stations, running with the default PFIFO queueing discipline installed on the wireless interface.

FQ-CoDel: As above, but using the FQ-CoDel qdisc on the wireless interface.

FQ-MAC: Kernel patched to include the FQ-CoDel based intermediate queues in the MAC layer (patching the mac80211 subsystem and the ath9k driver).

Airtime fair FQ: As FQ-MAC, but additionally including our airtime fairness scheduler in the ath9k driver.

Our evaluation is split into two parts. First, we validate the effects of the modifications in simple scenarios using synthetic benchmark traffic. Second, we evaluate the effect of our modifications on two application traffic scenarios, to verify that they provide a real-world benefit.

4.1 Validation of effects

In this section we present the evaluation of our modifications in simple synthetic scenarios designed to validate the correct functioning of the algorithms and to demonstrate various aspects of their performance.

4.1.1 Latency reductions

Figure 4 is the full set of results for our ICMP latency measurements with simultaneous TCP download traffic (of which a subset was shown earlier in Figure 2). Here, the FIFO case shows several hundred milliseconds of latency when the link is saturated by a TCP download. FQ-CoDel alleviates this somewhat, but the slow station still sees latencies of more than 200 ms in the median, and the fast stations around 35 ms. With the FQ-MAC queue restructuring, this is reduced so that the slow station now has the same median latency as the fast one does in the FQ-CoDel case, while the fast stations get their latency reduced by another 45%. The airtime scheduler doesn't improve further upon this, other than to alter the shape of the distribution slightly for the slow station (but retaining the same median). For this reason, we have omitted it from the figure to make it more readable.

For simultaneous upload and download the effect is similar, except that in this case the airtime scheduler slightly worsens the latency to the slow station, because it is scheduled less often to compensate for its increased airtime usage in the upstream direction. The graph of this case can be found in the online appendix.

4.1.2 Airtime usage

Figure 5 shows the airtime usage of the three active stations for one-way UDP traffic going to the stations. There is no reverse traffic and no contention between stations, since only the access point is transmitting data. This is the simplest case to reason about and measure, and it clearly shows the performance anomaly is present in the current Linux kernel (left half of the figure): The third station (which transmits at the lowest rate) takes up around 80% of the available airtime, while the two other stations share the remaining 20%.

The differences between the first two columns and the third column are due to changes in aggregation caused by the change to the queueing structure. In the FIFO and FQ-CoDel cases, there is a single FIFO queue with no mechanism to ensure fair sharing of that queue space between stations. So because the slow station has a lower egress rate, it will build more queue until it takes up the entire queueing space. This means that there are not enough packets queued to build sufficiently large aggregates for the fast stations to use the airtime effectively. The FQ-MAC queueing scheme drops packets from the largest queue on overflow, which ensures that the available queueing space is shared between stations, which improves aggregation for the fast stations and thus changes

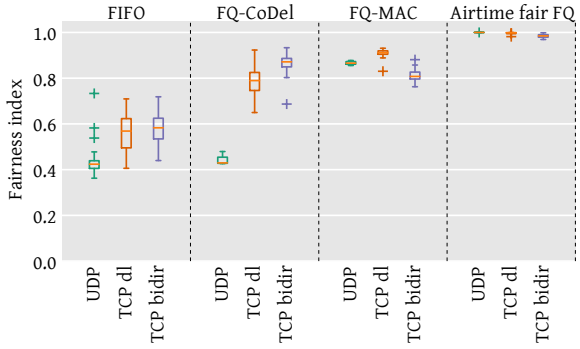


Figure 6: Jain's fairness index (computed over the airtime usage of the three stations) for UDP traffic, TCP download, and simultaneous TCP upload and download traffic.

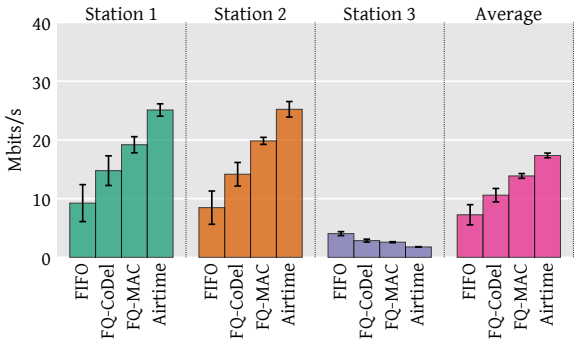


Figure 7: Throughput for TCP download traffic (to clients).

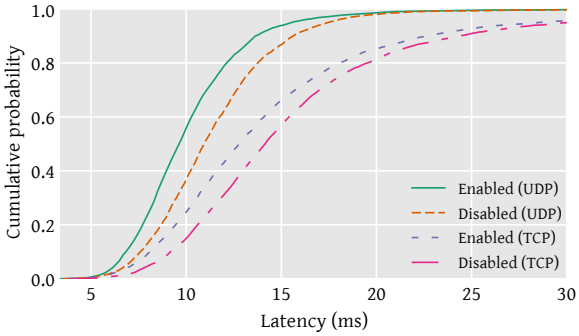


Figure 8: The effects of the sparse station optimisation.

the airtime shares. Referring back to Table 1, the values correspond well to those predicted by the analytical model. The fourth column shows the airtime fairness scheduler operating correctly – each station receives exactly the same amount of airtime in this simple one-way test.

Going beyond the simple UDP case, Figure 6 shows Jain's fairness index for the airtime of the four different schemes for UDP (for comparison) and both unidirectional (to the clients) and bidirectional (simultaneous up and down) TCP traffic. The same general pattern is seen with TCP as with UDP traffic: The performance anomaly is clear for the FIFO case, but somewhat lessened for the FQ-CoDel and FQ-MAC cases. The airtime fairness scheduler achieves close to perfect sharing of airtime in the case of uni-directional traffic, with a slight dip for bidirectional traffic. The latter is because the scheduler only exerts indirect control over the traffic sent from the clients, and so cannot enforce perfect fairness as with the other traffic types. However, because airtime is also accounted for received packets, the scheduler can partially compensate, which is why the difference between the unidirectional and bidirectional cases is not larger than it is.

4.1.3 Effects on throughput

As was already shown in Table 1, fixing the performance anomaly improves the efficiency of the network for unidirectional UDP traffic. Figure 7 shows the throughput for downstream TCP traffic. For this case, the fast stations increase their throughput as fairness goes up, and the slow station decreases its throughput. The total effect is a net increase in throughput. The increase from the FIFO case to FQ-CoDel and FQ-MAC is due to better aggregation for the fast stations. This was observed for UDP as well in the case of FQ-MAC, but for FQ-CoDel the slow station would occupy all the queue space in the driver, preventing the fast station from achieving full aggregation. With the TCP feedback loop in place, this lock-out behaviour is lessened, and so fast stations increase their throughput.

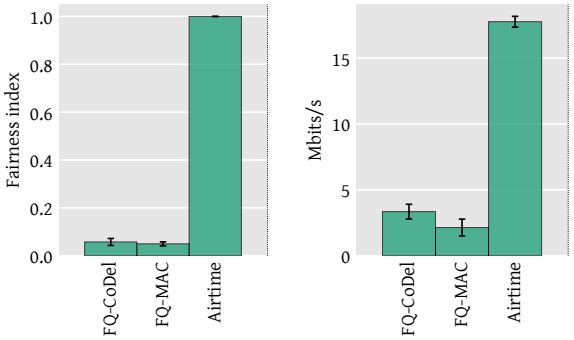
When traffic is flowing in both directions simultaneously, the pattern is similar, but with a slightly higher variance. The graph for the bidirectional case can be found in the online appendix.

4.1.4 The sparse station optimisation

To evaluate the impact of the sparse station optimisation, we add a fourth station to our experiments which receives only a ping flow, but no other traffic, while the other stations receive bulk traffic as above. We measure the latency to this extra station both with and without the sparse station optimisation. The results of this are shown in Figure 8. For both UDP and TCP download traffic, the optimisation achieves a small, but consistent, improvement: The round-trip latency to the fourth station is reduced by 10 to 15% (in the median) when the optimisation is in place.

4.1.5 Scaling to more stations

While the evaluations presented in the previous sections have shown that our modifications work as planned, and that they provide a substantial benefit in a variety of scenarios, one question is left unanswered – does the solution scale to more stations? To answer this, we arranged for



(a) Airtime usage fairness **(b)** Aggregate throughput

Figure 9: Aggregate results for the 30 stations TCP test.

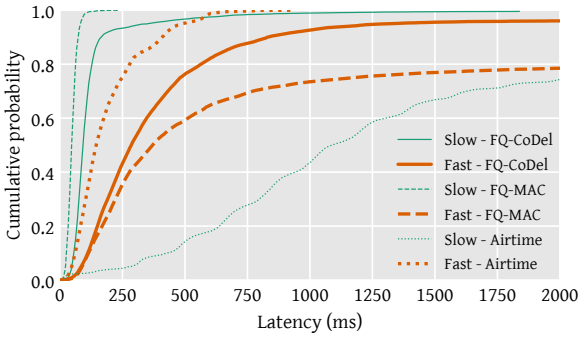


Figure 10: Latency for the 30 stations TCP test.

an independent third party to repeat a subset of our tests in their testbed, which features an access point and 30 clients. The nodes are all embedded wireless devices from a commercial vendor that bases its products on the OpenWrt/LEDE open-source router platform, running a LEDE firmware development snapshot from November 2016.

In this setup, one of the 30 clients is artificially limited to only transmit at the lowest possible rate (1 Mbps, i.e. disabling HT mode), while the others are configured to select their rate in the usual way, on a HT20 channel in the 2.4 Ghz band. One of the 29 "fast" clients only receives ping traffic, leaving 28 stations to contend with the slow 1 Mbps station for airtime and bandwidth.

In this environment, our downstream TCP experiment presented above was repeated, with the difference that each test was run for five minutes, but with only five repetitions, and without the FIFO test case. A subset of these results are shown in figures 9 and 10. From this experiment, we make several observations:

1. When the slow station is at this very low rate, it manages to grab around two thirds of the available airtime, even with 28 other stations to compete with. However, our airtime fairness scheduler manages to achieve completely fair sharing of airtime between all 29 stations. This is reflected in the fairness index as seen in Figure 9a.

2. As seen in Figure 9b, total throughput goes from a mean of 3.3 Mbps for the FQ-CoDel case to 17.7 Mbps with the airtime scheduler. That is, the relative throughput gain with airtime fairness is 5.4x in this scenario.

3. As can be expected, with the airtime fairness scheduler, the latency to the fast stations is improved with the increased throughput (Figure 10, green lines). However, the latency to the slow station increases by an order of magnitude in the median, as it is throttled to stay within its fair share of the airtime (Figure 10, dashed orange line). Overall, the *average* latency to all stations is improved by a factor of two (not shown on the figure).

4. With 30 stations, we see the sparse station optimisation being even more effective; in this scenario it reduces latency to the sparse station by a factor of two (not shown in the figures; see the online appendix).

Finally, we verify the in-kernel airtime measurement against a tool developed by the same third party that measures airtime from captures taken with a monitor device. We find that the two types of measurements agree to within 1.5%, on average.

4.2 Effects on real-world application performance

In the previous section we evaluated our solution in a number of scenarios that verify its correct functioning and quantify its benefits. In this section we expand on that validation by examining how our modifications affect performance of two important real-world applications – VoIP and web browsing.

4.2.1 VoIP

VoIP is an important latency-sensitive application which it is desirable to have working well over WiFi, since that gives mobile handsets the flexibility of switching between WiFi and cellular data as signal conditions change. To evaluate our modifications in the context of VoIP traffic, we measure VoIP performance when mixed with bulk traffic. As in Section 4.1.4 we include a virtual station as another fast station, and so these scenarios have three fast stations. Due to space constraints, we only include the case where the slow station receives both VoIP traffic *and* bulk traffic, while the fast stations receive bulk traffic. The other cases show similar relative performance between the different queue management schemes.

The QoS markings specified in the 802.11e standard can be used to improve the performance of VoIP traffic, and so we include this aspect in our evaluation. 802.11e specifies four different QoS levels, of which voice (VO) has the highest priority. Packets transmitted with this QoS marking gets both queueing priority and a shorter contention window, but cannot be aggregated. This difference

		5 ms		50 ms	
	QoS	MOS	Thrp	MOS	Thrp
FIFO	VO	4.17	27.5	4.13	21.6
	BE	1.00	28.3	1.00	22.0
FQ-CoDel	VO	4.17	25.5	4.08	15.2
	BE	1.24	23.6	1.21	15.1
FQ-MAC	VO	4.41	39.1	4.38	28.5
	BE	4.39	43.8	4.37	34.0
Airtime	VO	4.41	56.3	4.38	49.8
	BE	4.39	57.0	4.37	49.7

Table 2: MOS values and total throughput when using different QoS markings for VoIP traffic. Data for 5 ms and 50 ms baseline one-way delay.

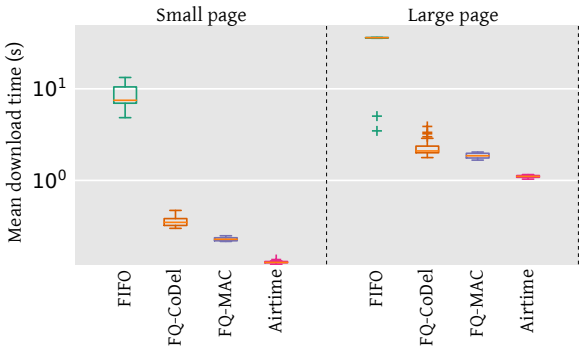


Figure 11: HTTP page fetch times for a fast station (while the slow station runs a bulk transfer). Note the log scale - the fetch time for the large page is 35 *seconds* for the FIFO case.

can dramatically reduce the latency of the traffic, at a cost in throughput because of the lack of aggregation. We repeat the voice experiments in two variants – one where the VoIP packets are sent as best effort (BE) traffic, and one where they are put into the high-priority VO queue. We also repeat the tests with a baseline one-way delay of both 5 ms and 50 ms.

To serve as a metric of voice quality, we calculate an estimate of the Mean Opinion Score (MOS) of the VoIP flow according to the E-model specified in the ITU-T G.107 recommendation [27]. This model can predict the MOS from a range of parameters, including the network conditions. We fix all audio and codec related parameters to their default values and calculate the MOS estimate based on the measured delay, jitter and packet loss. The model gives MOS values in the range from $1 - 4.5$.

Table 2 shows the results. This shows that throughput follows the trends shown in previous tests, as expected. Also as expected, the FIFO and FQ-CoDel cases have low MOS values when the voice traffic is marked as BE, and higher values when using the VO queue. However, both the FQ-MAC and airtime fairness schemes achieve *better* MOS values with best-effort traffic than the unmodified kernel does with VO-marked traffic. In the FQ-MAC and airtime cases, using the VO queue still gives a slightly better MOS score than using the BE queue does; but the difference is less than half a percent. This is an important improvement, because it means that with our modifications, applications can rely on excellent real-time performance even when unable to control DiffServ markings, as well as when the markings are removed in transit.

4.2.2 Web

Another important real-world application is web traffic. To investigate the impact of our modifications on this, we measure page load time (PLT) with emulated web traffic. Our test client mimics the common web browser behaviour of fetching multiple requests in parallel over four different TCP connections. We simply measure the total time to fetch a web site and all its attached resources (including the initial DNS lookup) for two different pages – a small page (56 KB data in three requests) and a large page (3 MB data in 110 requests). We run the experiments in two scenarios. One where a fast station fetches the web sites while the slow station runs a simultaneous bulk transfer, to emulate the impact of a slow station on the browsing performance of other users on the network. And another scenario where the slow station fetches the web sites while the fast stations run simultaneous bulk transfers, to emulate the browsing performance of a slow station on a busy network.

The results for the fast station are seen in Figure 11. Fetch times decrease from the FIFO case as the slowest to the airtime fair FQ case as the fastest. In particular, there is a an order-of-magnitude improvement when going from FIFO to FQ-CoDel, which we attribute to the corresponding significant reduction in latency seen earlier.

When the slow station is fetching the web page, adding airtime fairness increases page load time by $5 - 10\%$. This is as expected since in this case the slow station is being throttled. The graph for this can be found in the online appendix.

4.3 Summary

Our evaluation shows that our modifications achieve their design goal. We eliminate bufferbloat and the 802.11 performance anomaly, and achieve close to perfect airtime fairness even when station rates vary; and our solution scales successfully as more clients are added. We improve total throughput by up to a factor of five and reduce latency under load by up to an order of magnitude. We also achieve close to perfect airtime fairness in a scenario

where traffic is mixed between upstream and downstream flows from the different stations. Finally, the optimisation that prioritises sparse stations achieves a consistent improvement in latency to stations that only receive a small amount of traffic.

In addition, we show that our modifications give significant performance increases for two important real-world applications – VoIP and web traffic. In the case of VoIP, we manage to achieve better performance with best effort traffic than was achievable with traffic marked as Voice according to the 802.11e QoS standard. For web traffic we achieve significant reductions in page load times.

Finally, even though our evaluation scenario only features a limited number of stations, we have sought to represent a challenging scenario, by having high congestion rates and a large difference between the station rates. As such, we believe that it serves well as a validation of the effects. In addition, the feedback we have received from users of the code indicates that our solution works well in a variety of deployments.

5 Related work

There have been several previous studies on bufferbloat and its mitigations (e.g. [15, 29]), but only a few that deal with the problem in a WiFi-specific context. [10] and [15] both feature a WiFi component in a larger evaluation of bufferbloat mitigation techniques and show that while these techniques can help on a WiFi link, the lower-level queueing in the WiFi stack prevents a full solution of the problem in this space. [23] draws similar conclusions while looking at buffer sizing (but only mentions AQM-based solutions briefly). Finally, [4] touches upon congestion at the WiFi hop and uses different queueing schemes to address it, but in the context of a centralised solution that also seek to control fairness in the whole network. None of these works actually provide a solution for bufferbloat at the WiFi link itself, as we present here.

Several different schemes to achieve fairness based on modifying the contention behaviour of nodes are presented in [8, 12, 13, 19, 22, 30]. [12] and [19] both propose schemes that use either the 802.11e TXOP feature to allocate equal-length to all stations, or scaling of the contention window by the inverse of the transmission rate to achieve fairness. [13] develops an analytical model to predict the values to use for a similar scaling behaviour, which is also verified in simulation. [22] presents a modified contention behaviour that can lower the number of collisions experienced, but they do not verify the effect of their schemes on airtime fairness. [8] proposes a modification to the DCF based on sensing the idle time of the channel scaling CW_{min} with the rate to achieve fairness. Finally, [30] proposes a scheme for airtime fairness that runs several virtual DCF instances per node, scaling the number of instances with the rate and channel properties.

Achieving fairness by varying the transmission size is addressed in [5, 16, 20]. The former two predate the aggregation features of 802.11n and so [5] proposes to scale the packet size downwards by varying the MTU with the transmission rate. [20] goes the other way and proposes a scheme where a station will burst packets to match the total transmission length of the previous station that was heard on the network. Finally, [16] uses the two-level aggregation feature of 802.11n and proposes a scheme to dynamically select the optimal aggregation size so all transmissions take up the same amount of time.

Turning to schedulers, [7] and [6] both propose schedulers which work at the access point to achieve airtime fairness, the former estimating the packet transmission time from channel characteristics, and the latter measuring it after transmission has occurred. [21] proposes a solution for wireless mesh networks, which leverages routing metrics to schedule links in a way that ensures fairness. Finally, [17] proposes a scheme to scale the queue space at the access point based on the BDP of the flows going through the access point. Our solution is closest to [6], but we improve upon it by increasing accuracy and reducing implementation difficulty, while adding an important latency-reducing optimisation for sparse stations, as was described in Section 3.2.

A few proposals fall outside the categories above. [14] proposes a TCP congestion control algorithm that uses information about the wireless conditions to cap the TCP window size of clients to achieve fairness. Finally, there are schemes that sidestep the fairness problems of the 802.11 MAC and instead replace it entirely with TDMA scheduling. [3] proposes a scheme for TDMA scheduling in a mesh network that ensures fair bandwidth allocation to all connecting clients, and [28] implements a TDMA transmission scheme for infrastructure WiFi networks.

6 Conclusion

We have developed a novel two-part solution to two large performance problems affecting WiFi – bufferbloat and the 802.11 performance anomaly. The solution consists of a new integrated queueing scheme tailored specifically to eliminate bufferbloat in WiFi, which reduces latency under load by an order of magnitude. Leveraging the queueing structure, we have developed a deficit-based airtime fairness scheduler that works at the access point with no client modifications, and achieves close to perfect fairness in all the evaluated scenarios, increasing total throughput by up to a factor of 5.

Our solution reduces implementation complexity and increases accuracy compared to previous work, and has been accepted into the mainline Linux kernel, making it deployable on billions of Linux-based devices.

7 Acknowledgements

We would like to thank Sven Eckelmann and Simon Wunderlich for their work on independently verifying our implementation. Their work was funded by Open Mesh Inc, who also supplied their test hardware. We would also like to thank Felix Fietkau, Tim Shepard, Eric Dumazet, Johannes Berg, and the numerous other contributors to the Make-Wifi-Fast and LEDE projects for their insights, review and contributions to many different iterations of the implementation.

Portions of this work were funded by Google Fiber and by the Comcast Innovation Fund, and parts of the infrastructure was sponsored by Lupin Lodge.

8 References

[1] Adam Belay et al. 'IX: A Protected Dataplane Operating System for High Throughput and Low Latency'. In: *OSDI*. 11th USENIX Symposium on Operating Systems Design and Implementation. Oct. 2014, pp. 49–65.

[2] Teuku Yuliar Arif and Riri Fitri Sari. 'Throughput Estimates for A-MPDU and Block ACK Schemes Using HT-PHY Layer'. In: *Journal of Computers* 9.3 (Mar. 2014). doi: 10.4304/jcp.9.3.678-687.

[3] Naouel Ben Salem and Jean-Pierre Hubaux. 'A fair scheduling for wireless mesh networks'. In: *WiMesh*. 2005.

[4] Kan Cai et al. 'Wireless Unfairness: Alleviate MAC Congestion First!' In: *Proceedings of the Second ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization*. WinTECH '07. New York, NY, USA: ACM, 2007, pp. 43–50. doi: 10.1145/1287767.1287777.

[5] Joseph Dunn et al. 'A practical cross-layer mechanism for fairness in 802.11 networks'. In: *Broadband Networks, 2004. BroadNets 2004. Proceedings. First International Conference on*. IEEE, 2004, pp. 355–364.

[6] Rosario G. Garroppo et al. 'Providing air-time usage fairness in IEEE 802.11 networks with the deficit transmission time (DTT) scheduler'. In: *Wireless Networks* 13.4 (Aug. 2007), pp. 481–495. doi: 10.1007/s11276-006-9201-7.

[7] K. Gomez et al. 'On efficient airtime-based fair link scheduling in IEEE 802.11-based wireless networks'. In: *2011 IEEE 22nd International Symposium on Personal, Indoor and Mobile Radio Communications*. Sept. 2011, pp. 930–934. doi: 10.1109/PIMRC.2011.6140105.

[8] Martin Heusse et al. 'Idle Sense: An Optimal Access Method for High Throughput and Fairness in Rate Diverse Wireless LANs'. In: *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '05. New York, NY, USA: ACM, 2005, pp. 121–132. doi: 10.1145/1080091.1080107.

[9] Martin Heusse et al. 'Performance anomaly of 802.11 b'. In: *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*. Vol. 2. IEEE, 2003, pp. 836–843.

[10] Toke Høiland-Jørgensen, Per Hurtig, and Anna Brunstrom. 'The Good, the Bad and the WiFi: Modern AQMs in a residential setting'. In: *Computer Networks* 89 (Oct. 2015), pp. 90–106. doi: 10.1016/j.comnet.2015.07.014.

[11] T. Høiland-Jørgensen et al. *FlowQueue-Codel*. Internet Draft (informational). Mar. 2016.

[12] Li Bin Jiang and Soung Chang Liew. 'Proportional fairness in wireless LANs and ad hoc networks'. In: *Wireless Communications and Networking Conference, 2005 IEEE*. Vol. 3. IEEE, 2005, pp. 1551–1556.

[13] T. Joshi et al. 'Airtime Fairness for IEEE 802.11 Multirate Networks'. In: *IEEE Transactions on Mobile Computing* 7.4 (Apr. 2008), pp. 513–527. doi: 10.1109/TMC.2007.70740.

[14] K. Kashibuchi, A. Jamalipour, and N. Kato. 'Channel Occupancy Time Based TCP Rate Control for Improving Fairness in IEEE 802.11 DCF'. In: *IEEE Transactions on Vehicular Technology* 59.6 (July 2010), pp. 2974–2985. doi: 10.1109/TVT.2010.2048931.

[15] Naeem Khademi, David Ros, and Michael Welzl. 'The New AQM Kids on the Block: Much Ado About Nothing?' In: *Technical Report, Oslo University* (2013).

[16] Minho Kim, Eun-Chan Park, and Chong-Ho Choi. 'Adaptive Two-Level Frame Aggregation for Fairness and Efficiency in IEEE 802.11n Wireless LANs'. In: *Mobile Information Systems* 2015 (2015), pp. 1–14. doi: 10.1155/2015/548109.

[17] Dzmitry Kliazovich et al. 'Queue Management Mechanism for 802.11 Base Stations'. In: *IEEE Communications Letters* 15.7 (July 2011), pp. 728–730. doi: 10.1109/LCOMM.2011.051911.110642.

[18] M. Laddomada et al. ‘On the throughput performance of multirate IEEE 802.11 networks with variable-loaded stations: analysis, modeling, and a novel proportional fairness criterion’. In: *IEEE Transactions on Wireless Communications* 9.5 (May 2010), pp. 1594–1607. doi: 10.1109/TWC.2010.05.081191.

[19] Pochiang Lin, Wu-I. Chou, and Tsungnan Lin. ‘Achieving airtime fairness of delay-sensitive applications in multirate IEEE 802.11 wireless LANs’. In: *Communications Magazine, IEEE* 49.9 (2011), pp. 169–175.

[20] Tahiry Razafindralambo et al. ‘Dynamic packet aggregation to solve performance anomaly in 802.11 wireless networks’. In: *Proceedings of the 9th ACM international symposium on Modeling analysis and simulation of wireless and mobile systems*. ACM, 2006, pp. 247–254.

[21] R. Riggio, D. Miorandi, and I. Chlamtac. ‘Airtime Deficit Round Robin (ADRR) packet scheduling algorithm’. In: *2008 5th IEEE International Conference on Mobile Ad Hoc and Sensor Systems*. Sept. 2008, pp. 647–652. doi: 10.1109/MAHSS.2008.4660101.

[22] Luis Sanabria-Russo et al. ‘Future evolution of CSMA protocols for the IEEE 802.11 standard’. In: *Communications Workshops (ICC), 2013 IEEE International Conference on*. IEEE, 2013, pp. 1274–1279.

[23] A. Showail, K. Jamshaid, and B. Shihada. ‘Buffer sizing in wireless networks: challenges, solutions, and opportunities’. In: *IEEE Communications Magazine* 54.4 (Apr. 2016), pp. 130–137. doi: 10.1109/MCOM.2016.7452277.

[24] M. Shreedhar and G. Varghese. ‘Efficient fair queuing using deficit round-robin’. In: *IEEE/ACM Transactions on Networking* 4.3 (June 1996), pp. 375–385. doi: 10.1109/90.502236.

[25] T. Szigeti and F. Baker. *DiffServ to IEEE 802.11 Mapping*. Internet Draft (standards track). Nov. 2016.

[26] Godfrey Tan and John V. Guttag. ‘Time-based Fairness Improves Performance in Multi-Rate WLANs.’ In: *USENIX Annual Technical Conference, General Track*. 2004, pp. 269–282.

[27] *The E-model: a computational model for use in transmission planning*. Tech. rep. G.107. ITU-T, June 2015.

[28] Wim Torfs and Chris Blondia. ‘TDMA on commercial of-the-shelf hardware: Fact and fiction revealed’. In: *AEU-International Journal of Electronics and Communications* 69.5 (2015), pp. 800–813.

[29] Greg White. *Active Queue Management Algorithms for DOCSIS 3.0: A Simulation Study of CoDel, SFQ-CoDel and PIE in DOCSIS 3.0 Networks*. Tech. rep. Cable Television Laboratories, Inc., 2013.

[30] M. A. Yazici and N. Akar. ‘Running Multiple Instances of the Distributed Coordination Function for Air-Time Fairness in Multi-Rate WLANs’. In: *IEEE Transactions on Communications* 61.12 (Dec. 2013), pp. 5067–5076. doi: 10.1109/TCOMM.2013.111113.130120.

Persona: A High-Performance Bioinformatics Framework

Stuart Byma* Sam Whitlock* Laura Flueratoru† Ethan Tseng‡
Christos Kozyrakis§ Edouard Bugnion* James Larus*

Abstract

Next-generation genome sequencing technology has reached a point at which it is becoming cost-effective to sequence all patients. Biobanks and researchers are faced with an oncoming deluge of genomic data, whose processing requires new and scalable bioinformatics architectures and systems. Processing raw genetic sequence data is computationally expensive and datasets are large. Current software systems can require many hours to process a single genome and generally run only on a single computer. Common file formats are monolithic and row-oriented, a barrier to distributed computation.

To address these challenges, we built Persona, a cluster-scale, high-throughput bioinformatics framework. Persona currently supports paired-read alignment, sorting, and duplicate marking using well-known algorithms and techniques. Persona can significantly reduce end-to-end processing times for bioinformatics computations. A new *Aggregate Genomic Data* (AGD) format unifies sample data and analysis results, while enabling efficient distributed computation and I/O.

In a case study on sequence alignment, Persona sustains 1.353 gigabases aligned per second with 101 base pair reads on a 32-node cluster and can align a full genome in $\sim$16.7 seconds using the SNAP algorithm. Our results demonstrate that: (1) alignment computation with Persona scales linearly across servers with no measurable completion-time imbalance and negligible framework overheads; (2) on a single server, sorting with Persona and AGD is up to 2.3$\times$ faster than commonly used tools, while duplicate marking is 3$\times$ faster; (3) with AGD, a 7 node COTS network storage system can service up to 60 alignment compute nodes; (4) server cost dominates for a balanced system running Persona, while long-term data storage dwarfs the cost of computation.

*EPFL
†U. Politehnica of Bucharest *(work done during EPFL internship)*
‡Carnegie Mellon University *(work done during EPFL internship)*
§Stanford University

1 Introduction

In 2001, the approximate cost of sequencing a whole human genome was $100 million. In 2017, the cost of Whole Genome Sequencing (WGS) is rapidly approaching $100 [25], a faster-than-Moore's Law improvement. Low-cost sequencing is a key enabler of personalized medicine, which tailors treatments for patients to their genetic makeup, promising better diagnoses and more effective therapies.

The genomic data produced by modern sequencing machines, however, is unusable in its raw form. A large amount of pre-processing must be done before analysis. Depending on the sequencing parameters, raw data for one human cell genome can range from several gigabytes to hundreds of gigabytes. The data analysis and storage problems are already challenging and will continue to grow with the increasing ambition of doctors and researchers to sequence more humans and other organisms.

WGS processing consists of a number of steps, including read alignment (matching short snippets of genomic data against a known reference), sorting, indexing, duplicate marking and variant calling (determining where a patient has mutations/differences in their genome). For a typical human genome, processing reads and writes tens to hundreds of gigabytes of data and can require many hours with current tools. Computational costs were minor when sequencing was rare and expensive. However, as sequencing becomes an integral part of medical diagnosis and treatment, fast and efficient processing is invaluable for timely diagnosis and treatment.

Many existing tools run in parallel on a single multicore computer but are not designed to scale to server clusters or cloud computing (though there are significant efforts in this direction; see §7). A crucial challenge in scaling is that genomic data is stored in multiple file formats, none of which are appropriate for parallel or distributed computation. Sequencing machines produce raw genomic data in one file format (FASTQ [8]) while aligned data uses a different format (SAM/BAM [31]),

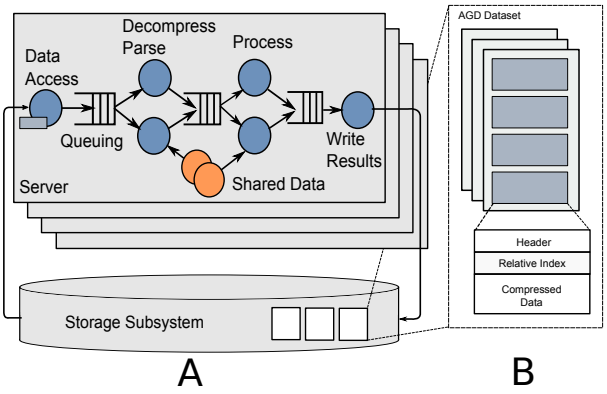


Figure 1: **A.**Persona architecture. Processing genomic data across multiple servers using a distributed dataflow framework. **B.**The Aggregate Genetic Data format stores data in columns to facilitate distributed processing.

which results in data duplication. Downstream analysis produces more files with different formats. In addition, common file formats are mainly row-oriented, which precludes efficient field access and frustrates data partitioning.

This state of affairs requires a new computing architecture to deal with the coming deluge of genomic data. We need a software architecture that runs effectively across computers ranging from a single machine to a cluster, so that genomic data processing can be performed in environments ranging from doctors' offices to hospitals and regional "gene banks".

To accomplish this, we require appropriate file formats that enable: (1) scalable, parallel access from multiple servers; (2) efficient use of both read and write bandwidth; (3) flexibility, to support the multiple phases in a genomics analytics pipeline; Additionally, scaling requires the efficient use of compute resources in terms of throughput and latency, which implies: (1) saturating compute resources of a server at all times, which requires data and task partitioning; (2) when possible, distribute computation across multiple servers; (3) scheduling this work, while avoiding stragglers [10]; (4) overlapping I/O with compute to hide latency.

In this paper, we present Persona, a scalable, high-performance framework for bioinformatics workloads, and the *Aggregate Genomic Data* (AGD) format. Figure 1 shows Persona and AGD at a high level. The goal of Persona and AGD is to provide a complete solution for bioinformatics computations, including (but not limited to) read alignment, sorting, duplicate marking, filtering, and variant calling. A secondary goal is extensibility — both Persona and AGD are designed and implemented in a way that allows straightforward integration of new capabilities (*e.g.,* different alignment algorithms or new data fields). Currently, Persona integrates well-known algorithms from the bioinformatics community, including those from BWA-MEM [30], SNAP [47], Samblaster [14], and samtools [31], so users can be confident in the results produced.

This paper makes the following contributions: (1) To address the limitations of disparate monolithic row-oriented files, the AGD format is a column-oriented file structure designed for compute, storage and I/O bandwidth efficiency, offering selective field and random access, distributed computation support, and unified storage of all genomic data for a given patient; (2) To run efficiently across single computers and moderate-sized clusters, we use distributed dataflow. Persona is built on Google TensorFlow [1], a state-of-the-art distributed dataflow framework. TensorFlow's coarse-grain dataflow minimizes framework overheads, yet, when augmented by a simple fine-grain mechanism, allows efficient use of all CPU resources in a cluster. We show that decoupling I/O granularity from task granularity in read alignment is necessary to maximize I/O bandwidth and balance work on modern multicore architectures; (3) We demonstrate linear scaling to the saturation point of our testbed storage cluster. We perform WGS alignment for a typical dataset in $\sim$16.7 seconds, a near order of magnitude improvement over existing solutions; (4) We demonstrate that the architecture is balanced from a total cost of ownership perspective, with the cost dominated by compute servers. Assuming full occupancy over 5 years, the cost of alignment is as little as 6.07¢. However, the long-term overall costs are likely to be dominated by storage.

Persona, AGD, and benchmarking scripts are freely available [13].

The rest of this paper is organized as follows: §2 provides some background in relevant algorithms and file formats. §3 describes the new AGD format and §4 describes the architecture of Persona. §5 evaluates our solution on a 32-server compute cluster attached to a scale-out storage subsystem. §6 provides some insight into bioinformatics workloads, and analyzes the TCO for different cluster options. Finally, we discuss related work in §7 and conclude in §8.

2 Background

The explosion of interest in and use of genomic data has been made possible by Next-Generation Sequencing (NGS) [6]. NGS machines, through a biochemical process called shotgun sequencing, read a genome by chopping long DNA strands into small pieces and reading these short snippets, which typically consist of 100 to 200 bases (A,T,C,G). The short snippets of a genome are called *reads* and must be *aligned* — reassembled into a full, coherent genome — before further analysis.

2.1 Bioinformatics Computations

Since our case study focuses on alignment, we provide some additional background.

To form a coherent genome, the reads in a raw dataset must be aligned to a reference genome (about 3 billion base pairs for a human). An *aligner* takes an individual read and attempts to find the most likely match in the reference sequence. Insertions, deletions, and mismatches between the bases are allowed, since genomes can have small mutations and the sequencing machines regularly misread base pairs. A read from a sequencing machine consists of three data fields: the bases (A,C,T,G or N, which is an ambiguous base), a quality score for each base indicating the machine's confidence, and metadata uniquely identifying the read. Datasets typically ensure that each base in the sample is overlapped by many reads — this is called *coverage* and is typically 30 to 50×. Raw datasets are typically single-ended, where each read is independent, or paired-ended, where reads are aligned as pairs with some gap between them. Reads are produced in arbitrary order.

Common algorithms for performing alignment include Smith-Waterman [43], an exact, dynamic programming algorithm, and BLAST (Basic Local Alignment Search Tool) [3], which uses seed-and-extend heuristics to locate short common words between sequences and extend them to reach a threshold. These approaches are expensive computationally, especially considering that modern read datasets with 50× coverage can contain billions of reads. Newer aligners, for example BWA-MEM [30], Bowtie [29], NovoAlign [37] and SOAP [32], rely on heuristics and algorithmic techniques such as tree-based indexing of the reference to speed up alignment. Others, such as SNAP [47], use hash-based indexing of the reference and are designed for multicore scalability. Alignment throughput is measured in bases aligned per second, a read-length agnostic measure.

Other expensive operations follow alignment. Downstream processing usually requires datasets to be sorted by read ID or aligned location in the genome. In addition, some downstream steps are more efficient with random access to the dataset. Sorting and indexing common data formats (§2.2) is often very time-consuming.

Once data is aligned, sorted and indexed, further filtering of data may take place. The preceding steps are usually followed by *variant calling*, another expensive process that compares the reassembled genome to the reference and attempts identify mutations. Common variant calling tools include GATK [34] and FreeBayes [16].

This is a sample of all the commonly used tools in bioinformatics; readers are referred to [38] for a more comprehensive survey.

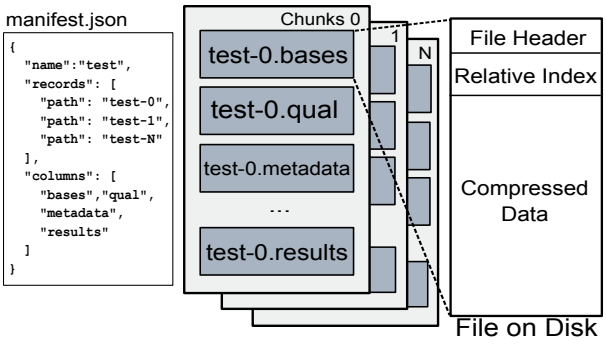


Figure 2: A dataset in AGD format.

2.2 File Formats

The canonical format produced by sequencing machines is FASTQ [8], an ASCII text format containing a delimited list of reads. FASTQ delimits reads by the `@` character, which makes parsing nontrivial as `@` is also an encoded quality score value. FASTQ files are usually distributed in a compressed format to save disk space.

The de facto standard for read and aligned data is the Sequence Alignment Map (SAM) format [31], or more often its binary, compressed version BAM. Variant calling results use the standard VCF format [9].

Typically, a dataset is stored in one FASTQ/SAM/BAM file, so these files are very large (50 to 100+ GB). While FASTQ just holds raw read data from a sequencer, SAM/BAM stores both the read and alignment data. The files are row oriented, so accessing a specific field requires reading all preceding entries, or generating a separate index file.

3 Aggregate Genomic Data Format

The *Aggregate Genomic Data* (*AGD*) format is a new extensible format for storing and manipulating genomic data designed to support the high I/O demands of Persona. AGD is designed for high-throughput read and write performance and to easily partition genomic data for parallel processing across one or more computers. Persona provides efficient utilities to export/import AGD to/from existing formats (SAM/BAM/FASTQ).

An AGD dataset is a table of records, each of which contains one or more fields (*i.e.,* a relational table). AGD stores the data in an indexed, column-store format (Figure 2). Record fields are stored by columns, which in turn are divided into large granularity *chunks* that reside in disk files. A descriptive manifest *metadata* file holds an index describing the columns, chunks, and records in an AGD dataset, in addition to other relevant data such as the names and sizes of contiguous reference sequences to which the dataset reads have been aligned. The manifest is implemented as a simple `JSON` file, which can be re-

constructed from the set of chunk files it describes. As an illustrating example, Persona uses three columns to store bases, quality scores, and metadata, and a fourth to store alignment results.

Operations on a genome dataset do not always require all elements in a record. For example, some duplicate marking schemes only require results, not base pairs or quality scores. In contrast to the row-oriented format of both FASTQ and SAM, each AGD column can be read independently and its data processed independently and simultaneously.

Moreover, AGD is extensible. A new record field (one or more columns) can be easily added by writing the column chunk files and adding appropriate entries to the metadata file. For example, Persona appends alignment results to a new AGD column. Any required parsing functions for a new column may be added to Persona. Columns can also be row-grouped, indicating that record indices align in those columns.

AGD columns are split into chunks containing varying number of records, enabling optimization for different storage subsystems. A chunk file contains a header, index, and data block (Figure 2). AGD specifies the record type in the chunk header, which informs applications how the data is stored (*e.g.,* what type of parsing to apply to each record). The index is relative, with offsets to records being generated by summing preceding index values. For more efficient random access, an absolute index can be generated on the fly.

AGD applies two techniques to reduce the size of the dataset: block compression of the data block and base compaction. The type of compression may be selected on a column-by-column basis. For example, a user may compress the bases column with *gzip* while using LZMA for the metadata. This flexibility allows tradeoffs between compressed file size and decompression time, which allow a user to balance the frequency of access against the size of a column. Our implementation uses *gzip*, as it has a good compression without being too compute-intensive. An additional optimization of *base compaction* is applied to the base reads column, which stores base characters using 3 bits each, with 21 bases in a 64-bit word.

The choice of chunk size is an important factor to maximize I/O performance. Larger chunk sizes have better compression ratios and lower overhead due to large contiguous reads from local storage. However, smaller chunk sizes decrease the I/O and decompression latency during which processing cores may stand idle.

4 System Architecture

We use a coarse-grain dataflow execution model for Persona. The major functions of the system — I/O, computation, and system management — are separated into

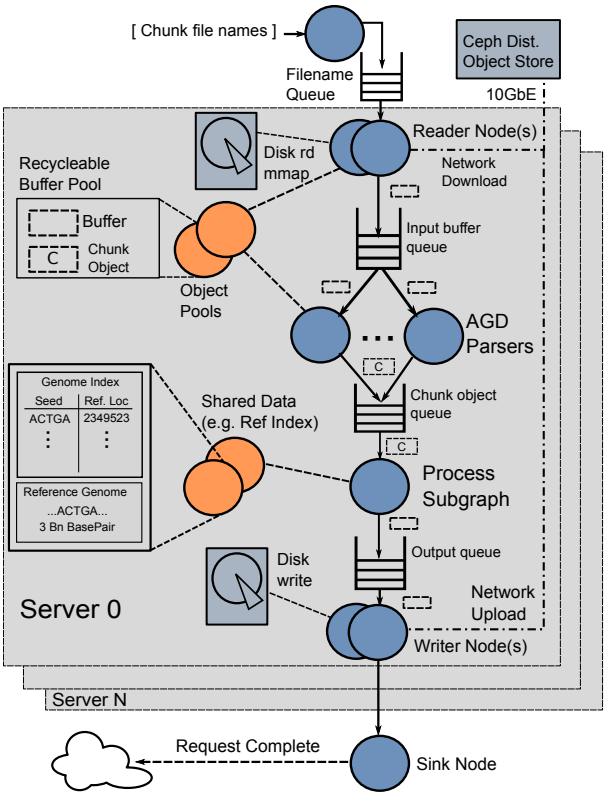


Figure 3: Persona dataflow architecture.

dataflow kernels. Each kernel can be mapped to available hardware resources (servers, cores, threads, or accelerators). This computation model simplifies the design, implementation, and deployment of the system, and allows for simple integration of new processing steps. In particular, the explicit flow between kernels simplifies performance and bottleneck analysis and makes it easy to adjust queuing for flow control and load balancing. Dataflow semantics mean that independent tasks always execute in parallel, both at the multicore and server levels.

We use Google TensorFlow as our underlying dataflow execution engine [1]. Although designed for machine learning, the core of TensorFlow is a generic dataflow engine. In TensorFlow, dataflow operators are called *nodes*, which are assembled into computation *graphs* using a Python API. Underlying *kernel* implementations of nodes are written in C++ and compiled alongside the runtime framework.

We demonstrate that it is possible to use the Google engine in a different context with minimal overhead (1%). To achieve this low overhead, Persona: (1) Uses a coarse-grain dataflow execution model between kernels, while adding a *fine-grain* execution model within compute-intensive kernels; (2) Uses pools of reusable objects to buffer data and implement a zero-copy archi-

tecture; (3) Controls memory usage by limiting the size of object pools and the length of the queues between kernels; (4) Balances the parallelism of I/O and alignment to keep all CPU threads busy.

4.1 Persona Architecture

Persona consists of two layers: a set of TensorFlow dataflow operators that read, parse, write, and operate on AGD chunks, and a thin Python library that stitches these nodes together into optimized subgraphs for common I/O patterns and bioinformatics functions.

Figure 3 shows an instance of a Persona graph on a single server. AGD chunk file names are consumed by reader nodes that read data from disk or network sources. AGD parsers decompress read chunks, enqueuing them for the *process* subgraph. The *process* subgraph contains the compute-intense operations — alignment, sorting, duplicate marking, variant calling, etc. The writer nodes store results from the process stage. Shared data objects and pools provide recyclable buffers for AGD chunks and results, and other shared objects such as the multi-gigabyte reference indexes required for some aligners.

Individual dataflow nodes and queues can be stitched together using the Python API however the user desires. However, certain configuration patterns are more efficient. The following subsections describe subgraphs that Persona uses to achieve high performance.

4.2 I/O Input Subgraph

The input subgraph is designed to keep the process subgraph fed with data while incurring minimal overhead. Reader nodes are implementations that read AGD chunks from storage. Currently, Persona supports a local disk or the Ceph object store [46] — other storage systems can be supported simply by writing the interface into a new Reader dataflow node. For disk files, Reader nodes `mmap` AGD chunk files, producing a handle to a read-only mapped file memory region. For network files, Reader nodes request the chunk files from a storage system (*e.g.,* Ceph), putting each into a recyclable buffer obtained from a buffer Pool. Once a chunk has been read, it passes via a queue to an AGD Parser node, which decompresses and parses the chunk into a useable, in-memory *chunk object*. Chunk objects are then passed to the process subgraph via a central queue.

4.3 Process Subgraphs

Process subgraphs implement the bioinformatics operations on the AGD chunk objects. We describe the implementation of several major functions and variants that are currently implemented in Persona. In our experience, since the I/O and parallel execution are provided by Persona, integrating existing tools is usually simple.

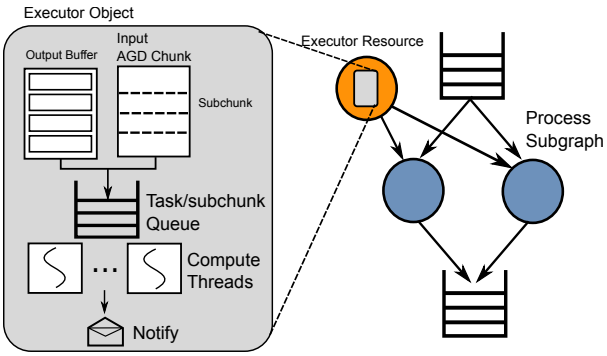


Figure 4: To abstract and share threads in a coherent way between parallel compute-intense kernels, a thread-owning executor object is provided via a resource.

SNAP Alignment The Persona SNAP aligner node uses the SNAP short read aligner [47], an open source tool that is highly optimized for modern servers with a large amount of memory and many cores. To attain maximum performance, each core in the system should be running the SNAP algorithm continuously on AGD chunks, however we found the granularity of AGD chunks, being optimized for storage, is too coarse for threads and produces work imbalance that leads to stragglers. To remedy this, execution of the alignment algorithm is delegated to an *executor* resource that owns all of the threads, and implements a fine-grain task queue (Figure 4). Multiple parallel aligner nodes then feed chunks to this *executor*, and wait for them to be completed. All cores in the system are thus kept running continuously doing meaningful work.

When executed, the aligner node receives chunk objects containing reads (base pairs and quality scores), a handle to a buffer pool of output objects, and a handle to the *executor* resource. The chunk object and output buffer are logically divided into subchunks and placed in the *executor* task queue as (subchunk, buffer) pairs. Once a full chunk is completed, the originating aligner node is notified, and the result buffer is placed in the subgraph output queue.

BWA-MEM Alignment BWA-MEM [30] is another popular read alignment tool that uses the Burrows-Wheeler transform to efficiently find candidate alignment positions for reads. We integrate BWA-MEM in the same manner as SNAP, using the *executor* resource with a fine-grain task queue (Figure 4). We call BWA-MEM alignment functions directly, with only several lines of cosmetic code changes required. For single-read alignment, this approach is straightforward, however for

paired reads, BWA-MEM incorporates a single-threaded step over sets of reads to infer information about the data. This leads to better alignment results, but separates the computationally intense multithreaded step into two parts. Therefore, the *executor* resource for BWA paired alignment divides the system threads among these tasks. We find a balance empirically, but because the computation times are data dependent, some efficiency is lost.

Sorting and Duplicate Marking Persona also integrates full dataset sorting by various parameters, including mapped read location and read ID. The sort implementation is a simple external merge sort, where several chunks at a time are sorted and merged into temporary file "superchunks". A final merge stage merges superchunks into the final sorted dataset. Persona sort is several times faster than samtools sorting of SAM/BAM files (§5).

Duplicate marking is a process of marking reads that map to the exact same location on the reference genome. This step is often performed since duplicate data can disrupt downstream statistical methods. Persona duplicate marking uses an efficient hashing technique based on the approach used by Samblaster [14].

4.4 I/O Output Subgraph

The output subgraph mirrors the input subgraph, with Writer nodes writing AGD chunks to disk or a Ceph object store, with an optional compression stage. In general, the process subgraph is responsible for ensuring AGD chunks to write are properly formatted for a given AGD column, as the Writer nodes are generic.

Persona also implements an output subgraph for the common SAM/BAM format for compatibility with tools that have not been integrated or do not yet support AGD.

4.5 Memory Management and Queuing

Proper memory management is necessary to efficiently use the underlying server hardware. In particular, it is important to avoid freeing, reallocating, and copying memory and to avoid bringing in too much data, which sits idle, or too little data, which stalls the pipeline.

We avoid using TensorFlow tensors directly for storing data, as they are not amenable to byte strings or raw buffers. Instead, we pass tensors of *handles*, which are identifiers for *resources* stored in the TensorFlow Session. The resources in Persona are the pools and their objects (buffers, chunk objects, shared read-only objects) as shown in Figure 3. With this technique, Persona performs no unnecessary copies.

Because computations in bioinformatics tend to be compute- or memory-bound, the input subgraph generally runs ahead of the alignment subgraph, quickly filling the process subgraph input queue. Persona controls memory pressure by limiting the queue length and therefore the number of objects passed around. The total quantity of objects is the sum of the queue lengths and the number of dataflow nodes that use an object. Overall memory use in Persona is stable after the input queues are filled. Because of the relatively coarse granularity of AGD chunks, default queue lengths are set to the number of parallel downstream nodes they feed, but can be tuned lower for low-memory systems.

Queue capacity is kept at a level that ensures there is always data to feed the process subgraph, but the individual servers do not have too many AGD chunks in their pipelines, which can lead to stragglers. A server can become a straggler if its queue contains "expensive" chunks with high compute latency. Work stealing [5] is an alternative to avoid stragglers, but the approach of bounding the queues is simpler and incurs less communication in a distributed system.

4.6 Discussion

Using TensorFlow as a general dataflow engine was a key design decision that had many benefits, but also led to some challenges. Bioinformatics data is not particularly amenable to storage in tensors. Initially, we had stored strings of bases, qualities and metadata in `string` type tensors, however this led to large amounts of small memory allocations, and constant data copying since the `std::string` type owns its data. This prompted the decision to move to the recyclable buffer pooling strategy outlined in the previous subsections. In an ideal world, the dataflow engine and runtime of TensorFlow would be separate from the Tensor data type and allow arbitrary types.

The execution semantics of TensorFlow also caused some issues when trying to maximize performance, especially in the multithreaded aligner kernels. Because graphs are executed in steps, there is necessarily a delay between one execution of a kernel and the next. Therefore, parallelism must be used in the graph to ensure that threads do not sit idle between executions. However, ad-hoc sharing of threads between these multiple kernels via the TensorFlow CPU device threadpool becomes difficult due to the way we need to split AGD chunks to reduce thread-level stragglers. The solution to this was the method described in §4.3, where all threads executing a given task are owned by a shared resource that can be fed with work by multiple kernels.

Despite these difficulties, we were still pleased overall with TensorFlow. The framework provides numerous features that greatly ease development and optimization, such as node-level profiling, graph visualization, and runtime statistics including current queue states or any other variable one wishes to track. We were also

pleasantly surprised at how seamlessly the implementation was able to overlap disk or network I/O with computation. We also found that the dataflow semantics in general enforce a fairly high degree of code separation and modularity, which makes for seamless extension for new support (*e.g.,* different I/O subsystems).

5 Evaluation

5.1 Experimental Setup

We use a cluster of typical datacenter machines, each with two Intel Xeon E5-2680v3 CPU chips at 2.5GHz and 256 GBytes of DRAM. With 12 cores per socket nd hyperthreading enabled, each node has 48 logical cores. All machines run Ubuntu 16.04 Xenial Linux. Each machine includes 2 SSDs in RAID1 configuration for the OS, 6 SATA disks (4TB, 7200 RPM, 6 GB/s), a hardware RAID controller, and 10GbE network interface. For single-node (local) experiments, we store the input data on a 20 TB RAID0 disk array. For distributed (cluster) experiments, we store the AGD dataset in a Ceph distributed object store [46] spread over 7 servers. The Ceph cluster is configured to use 3-way replication and each of its 7 nodes has 10 disks. The compute and storage are connected by a 40GbE-based IP fabric consisting of 8 top-of-rack switches and 3 spine switches.

Persona accesses Ceph objects via the Rados API. Using the `rados bench` tool, we measure the peak Ceph read throughput of our configuration at 6 GB/s, with sequential reads and evenly distributed data.

In all our experiments, we use half of a paired-end whole genome dataset from Illumina [12] (ERR174324), which consists of 223 million single-end 101-base reads, and is 18 GB in gzipped-FASTQ format and 16 GB in AGD format. The use of single-end read data is an arbitrary choice; Persona's integrated aligners and AGD also support paired-end alignment. The reference genome to which we align the dataset is the common hg19 human genome [23]. As mentioned in §2, alignment throughput is measured in bases aligned per second.

5.2 Persona Configuration

All execution uses the TensorFlow direct session, unmodified. For cluster-wide execution, Persona launches a TensorFlow instance per compute server. Within each server, the first stage in the TensorFlow graph fetches a chunk name from the manifest server; the latter is implemented as a simple message queue. Unless noted, the AGD chunk size is 100,000, grouped into 2231 chunks. At this chunk size, both the bases and the qualities are ∼3.5 MB. As our performance analysis focuses mainly on alignment, we read only these two columns of each chunk, totaling ∼7 MB per chunk.

	SNAP	AGD Single Node	Speedup
Disk(Single)	817 sec	501 sec	1.63
Disk(RAID)	494 sec	499 sec	0.99
Network	760 sec	493.5 sec	1.54
Data Read	18GB	15GB	1.2
Data Written	67GB	4GB	16.75

Table 1: Dataset Alignment Time, Single Server

5.3 I/O Behavior of AGD

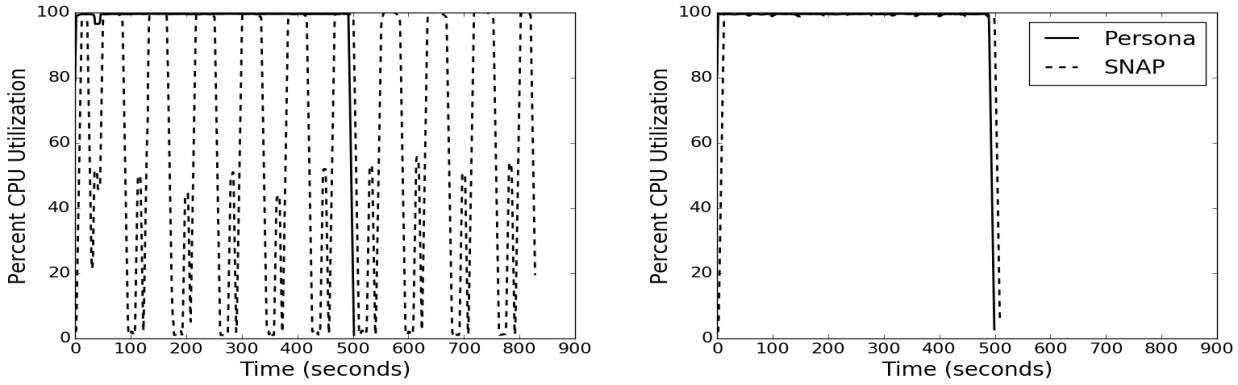
We first study the I/O behavior of Persona and AGD. I/O behavior in Persona is fundamental, since we can never assume a given patient's genome data will already be in memory (or that it even fits in memory). We perform alignment using different disk I/O configurations, using the SNAP alignment subgraph and comparing to the SNAP standalone program. We use SNAP instead of BWA because it has higher throughput and is better able to exercise the I/O subsystem. The *single disk* configuration stores the genome (and the results) on a single local disk. The *RAID0* configuration uses a hardware RAID0 array of 6 disks to increase bandwidth. Both SNAP and Persona are tuned for best performance, and use 47 aligner threads.

Figure 5 provides a characterization of the CPU utilization using a single disk and the full RAID0 configuration. Both systems overlap I/O and decompression with alignment: SNAP uses an ad-hoc combination of threads, whereas Persona leverages dataflow execution. Figure 5a and Figure 5b show that Persona is CPU bound in both configurations, but that SNAP can only use the CPU resource fully in the RAID0 configuration.

In particular, Figure 5a shows a cyclical pattern with SNAP where the operating system's buffer cache write-back policy competes with the application-driven data reads; during periods of writeback, the application is unable to read input data fast enough and threads go idle.

Table 1 summarizes the difference in terms of the amount of I/O traffic required as well as the impact on execution time. While the column-orientation of AGD has a marginal benefit in terms of data input, it has a 16.75× impact on data output, and a 1.63× speedup for the single-disk configuration. When the storage subsystem provides sufficient bandwidth, as for the RAID0 configuration, the performance of SNAP and Persona are nearly identical. Persona, however, does at least the same amount of work with less hardware and eliminates the disk I/O bottleneck.

The benefits of column-orientation of AGD are not limited to local disks. Table 1 also shows the speedup of 1.54× when the data is stored on Ceph network-attached



(a) Single Disk CPU

(b) RAID 0 CPU

Figure 5: Comparison of SNAP (GZIP'd FASTQ) and Persona (AGD) in CPU utilization with single disk and RAID0.

storage[1].

Finally, Table 1 shows that, by overlapping I/O with computation in meaningful-sized pieces, the performance of Persona is nearly identical to SNAP and CPU bound in three very different storage configurations.

5.4 Single-node CPU Alignment

We characterize the thread scaling behavior for Persona in both the SNAP and BWA-MEM aligners, while comparing them to their standalone baselines, with single-end alignment. These experiments show that Persona imposes negligible core-scaling overhead on the subsystems we have integrated, and avoids thread and I/O saturation issues by efficient overlapping.

Figure 6 shows the scalability of standalone SNAP and BWA-MEM compared to Persona as a function of the number of provisioned aligner threads on the 48 core server. The experiments were measured on the RAID0 configuration so that SNAP has enough I/O bandwidth. For SNAP, Figure 6 shows clearly: (1) a near-linear speedup for up to 24 threads, corresponding to the 24 physical processor cores of the server; (2) that, beyond 24 cores, the 2nd hyperthread increases the alignment rate of a core by 32%. At 48 threads however, contention with I/O scheduling causes a drop in performance in SNAP. Persona is less sensitive to operating system kernel thread scheduling decisions because of TensorFlow's built-in queue abstractions.

BWA scales fairly well to 24 threads, but afterwards suffers from high memory contention after hyperthreading kicks in, something we can not fix without significant changes to the codebase. However, because Persona avoids setting up and tearing down threads for different steps of processing, Persona's BWA-MEM subgraph scales slightly better with more threads than the standalone program.

[1]SNAP does not natively support reading from Ceph, so we use the `rados` utility to pipe the dataset in gzipped FASTQ format, and pipe the resulting SAM file into Ceph.

5.5 Cluster Scalability

Figure 7 shows the throughput of two different systems as a function of the number of nodes. "Actual" represents the measured performance of Persona using the SNAP alignment node, reported in gigabases aligned per second for a single genome (*i.e.,* a measurement of latency). "Simulation" is the ideal speedup line based on the maximum local server performance of ∼45.45 megabases aligned per second (see §5.4).

Persona scales linearly up to the available 32 nodes by making efficient use of all compute resources, hiding all I/O latencies and addressing the straggler problem through shallow queues. Again, we use SNAP because the higher throughput is better able to exercise the I/O subsystems. When considering BWA-MEM, alignment throughput may be lower per node, but may scale to higher numbers of servers. We reiterate that our point is not to compare BWA-MEM to SNAP, but to show that Persona is able to scale to a high number of servers while keeping process subgraphs fully supplied with data.

Using 32 servers and the SNAP process subgraph, Persona aligns the genome in 16.7 seconds, from the beginning of the request to when all results are written back to the Ceph cluster. This corresponds to 1.353 gigabases aligned per second. As far as we are aware, this represents the fastest whole genome alignment to date.

We use a different methodology to test the scalability of the storage cluster. For this, we deploy multiple "virtual" TensorFlow sessions per server and replace the CPU-intensive SNAP algorithm with a stub that simply suspends execution for the mean time required to align a chunk, and then output a representative (but obviously

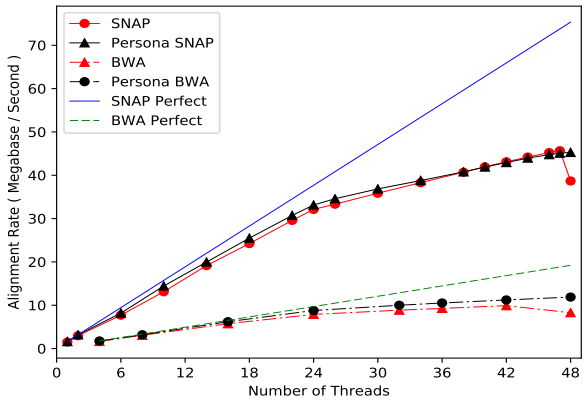


Figure 6: Throughput scaling across cores. Persona adds no measurable overhead.

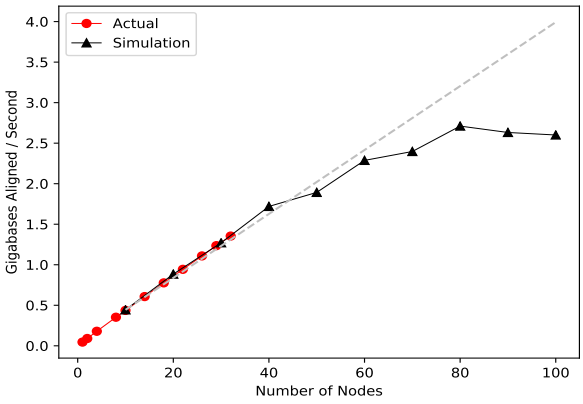


Figure 7: Actual cluster throughput up to 32 servers using the Persona SNAP aligner. Simulated throughput while scaling to 100 servers.

Tool	Time	Speedup
Persona	556 sec	1.0×
Samtools	856 sec	1.54×
Samtools w/ conversion	1289 sec	2.32×
Picard	2866 sec	5.15×

Table 2: Dataset Sort Time in Seconds, Single Server

incorrect) result.

Figure 7 shows the results in the "Simulation" line. We first validate that the simulation matches the "Actual" measurements up to 32 nodes. We then observe that the Ceph cluster scales to ∼60 nodes without loss of efficiency. Beyond 60 nodes, and for an AGD chunk size of 100,000 reads, write performance of the alignment results limits performance.

5.6 Sorting and Duplicate Marking

We also compare Persona in sorting performance to Samtools [31] and Picard [27], standard utilities for sorting SAM/BAM files. Table 2 shows the results when configuring Samtools to use all 48 cores available. Picard does not have an option for multithreading. Samtools requires sorting input in BAM format; we include both sort and sort + conversion times. Persona can directly process aligned results in AGD, performing up to 2.32 times faster than Samtools when considering the file conversion time. Persona's sort implementation is currently naive, using `std::sort()` across chunks, and we believe these results can be improved substantially.

We compare Persona's duplicate marking performance to Samblaster [14], whose algorithm we have used in our implementation. Samblaster can mark duplicates at 364,963 reads per second, while Persona, which uses Google's optimized dense hashtable, can mark duplicates at 1.36 million reads per second. Note that Persona also uses less I/O since only the results column needs to be read/written from the AGD dataset.

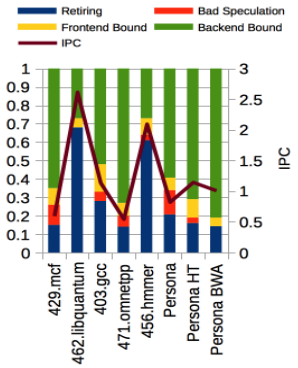
5.7 Conversion and Compatibility

To support existing sequencer output formats and other tools that have not yet been integrated, Persona can import FASTQ and export BAM formats at high throughput. FASTQ is imported to AGD at 360 MB/s, while BAM format files are produced from AGD at 82 MB/s.

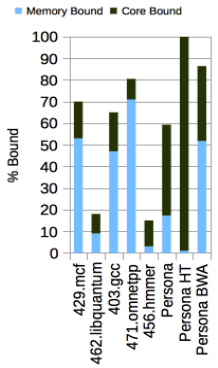
6 Discussion

Our performance analysis focuses on alignment, as it is the most compute-intense step we have yet integrated into Persona. As this is a primary bottleneck for analysis, we used Intel's VTune Amplifier [41] to profile both BWA-MEM and SNAP while running in Persona, to identify any possible avenues for improvement. Figures 8a–8b summarize our findings, while comparing to several relevant SPEC benchmarks.

Both aligner profiles share some similarity, in that they are heavily CPU backend-bound (*i.e.,* many cycles stalled due to lack of resources for accepting μOps into the pipeline, like D-cache misses or occupied functional units). With SNAP, we see that the issue is due to the core and not memory access — this is due to short but frequent calls to a local alignment edit distance function that has a small instruction mix and many data dependent instructions and branches. In BWA-MEM, the system is much more memory bound. VTune reports that this is due mostly to cache misses and DTLB misses, and our findings corroborate previous analyses [48]. This also helps explain our improved thread scaling — by restricting primary functions to sets of cores, we reduce thread



(a) Analysis breakdown.



(b) Core and memory bound levels.

Figure 8: Workload analysis (with and without Hyper-threading) compared to several SPEC benchmarks.

interference in the memory hierarchy.

6.1 TCO of Cluster Architectures

Personalized medicine has become practical because of dramatic decreases in the cost of genome sequencing. In light of these decreases, it is worth considering the cost contributions of storage and computation. We consider three cases: a single system attached to a single NGS sequencer, our own balanced cluster, and a nation-wide solution. We limit the analysis to alignment, the most expensive computation we have yet integrated into Persona.

First, Figure 5 shows the performance of single server, where genomic data is stored, aligned, and processed on a local machine. A single server can align ∼144 full sequences per day. Considering the total cost of ownership (TCO) of the server over 5 years, this implies a cost of 4.1¢ per alignment, assuming full utilization. Note that this scenario has limited genome storage capacity.

Second, there are economies of scale for sequencing, and a more likely scenario would be a regional center providing sequencing and processing services. A small cluster and network storage subsystem, as we have used in our experiments, could support 5173 alignments per day. Figure 7 shows that our storage cluster can sustain the I/O requests of a cluster of twice this size, offering expansion capacity. Table 3 summarizes the cluster compute and storage costs over a 5 year lifetime. For the network fabric cost, we determine the per-port cost of the 8-TOR, 3-spine architecture deployed in our physical cluster, and multiply by the number of ports used. Table 3 shows that, assuming the system is fully loaded, the TCO of a genome alignment on such a regional cluster is 6.07¢, higher than above because of the larger storage subsystem needed to support the throughput.

Item	Unit cost	Units	Total
Compute Server	$8,450	60	$507K
Storage server	$7,575	7	$53K
Fabric ports	$792	67	$53K
Total			$613K
TCO(5yr) [21]			$943K
Cost/Alignment (100% Utilization)			6.07¢

Table 3: Cluster TCO and alignment costs. The storage cluster has 126 TB of usable capacity, corresponding to approximately 6,000 sequenced genomes.

Third, a nation-wide solution would be needed to support initiatives such as Genomics England's 100,000 Genomes [17]. For this, additional storage is required as our balanced cluster has a usable capacity of 126 TB, which can store 6,000 in AGD format (1 days worth of sequencing). One can use the 60:7 ratio of compute to storage machines as a "not to exceed" scaling guide. The TCO model of Table 3 can be adjusted to estimate the capacity and throughput requirements of a deployment.

Storage is the dominant cost of a cluster and of genome processing. With our current high-throughput storage subsystem, the cost per genome for storage is $8.83, two orders of magnitude higher than the alignment cost. Genomes that are not being actively processed could be stored in tiered storage system using slower, lower-cost storage and erasure coding [11]. Currently, using Amazon Glacier storage ($0.007 GB/month [4]), a full genome could be stored for 5 years for $6.72, only slightly less expensive than locally hosted storage. Note that with higher coverage datasets, storage amounts and cost would increase.

Computation is far from the dominant contribution to the cost of sequencing a genome. Storage, while more expensive, is still far from a significant expense, but if the cost of sequencing continues to decline at its faster-than-Moore's-Law rate, storage may become the limiting factor in widespread genome sequencing. Novel compression for genomic data, such as reference-based compression [15], will likely be required.

7 Related Work

Because of its potential, bioinformatics and genomics have been the topic of much research. Large organizations such as the Broad Institute have established pipelines (Genome Analysis Toolkit [34]), a system similar to Persona. GATK also employs sharding for parallel data access (i.e. HDFS), but uses the standard SAM/BAM formats, often merging multiple input files into single files, which can limit scalability. Recently,

GATK has also been ported a cloud environment, Google Genomics [26]. Microsoft also advertizes cloud-based genomics capabilities [35]. However, these companies have not released details of their internal systems architectures, so it is unclear how they compare.

In terms of file formats, the recent ADAM format [33] is most similar to AGD. It also uses a column store format to achieve better compression. In addition, data is serialized using a common framework (Avro) that supports multiple languages and is easily parsed. ADAM relies on Spark and HDFS for distributed computation, again restricting users to a single storage subsystem type. In terms of performance, ADAM claims a $\sim 2\times$ speedup over Picard in single node sorting, whereas Persona achieves a $\sim 5\times$ speedup. HDF5 [44] is a general purpose hierarchical file format that can also support a bioinformatics schema similar to ADAM. In contrast to AGD, it restricts users to MPI for multiprocessing and is difficult to tune for high performance. TileDB [39] is a system that stores multi-dimensional array data in fixed size data tiles, similar to HDF5 but superior in write performance and concurrency. TileDB "chunking" is similar to AGD, but it employs a more rigid data model and is generally much more complex. Parallel access is implemented using MPI as in HDF5. Futhermore, GenomicsDB [22] is built on TileDB to store genomic variant data in 2D arrays, columns and rows correspond to genome positions and samples, respectively.

AGD differs substantially from these formats in that it is simple and requires only a way to store keyed chunks of data. The AGD API to access chunk data can simply be layered on top of different storage or file systems, using those system's APIs for parallel access, distribution, replication, etc.

Distributed alignment has been explored before, for example CloudBurst [42], which uses Hadoop MapReduce. They also find that the problem scales linearly and that distribution can result in significant speedups. CloudBurst reports 7 million reads aligned to one human chromosome in 500 seconds using 96 cores (5256 bases aligned per second per core), however a direct performance comparison is difficult because the alignment algorithm is different, the read size is different (36 base pairs versus our 101), and the cluster architecture and CPU were different. Cloud-Scale BWAMEM [7] is a distributed aligner that can align a genome in $\sim$80 minutes over 25 servers, but requires different file formats for single (SAM) or distributed computation (ADAM). SparkBWA [2] is similar, scaling alignment out over a Spark cluster, but not achieving linear scaling. ParSRA [19] shows close to linear scaling using a PGAS approach, but relies on FUSE to split input files among nodes. Eoulsan [28] uses MapReduce to perform several pipeline steps and supports different aligners. Pmap [24] uses MPI to scale several different aligners across servers and claims linear scaling.

Other efforts include SAND [36], where alignment is divided into stages for reads, candidate selection and alignment on dedicated clusters using algorithms similar to BLAST. There have also been efforts to distribute BLAST computation itself [40]. Others have shown that aligning reads to a reference genome scales linearly [20]. merAligner [18] implements a seed-and-extend algorithm that is highly parallel at all stages, but uses fine-grained parallelism more amenable to supercomputing systems rather than the clusters or datacenters that Persona targets. GENALICE Map [45] reports 92 million bases aligned per second on a single machine, faster than even SNAP, however it is a closed-source proprietary product.

In contrast to previous work, Persona and AGD provide a *general* high-performance framework that facilitates *linear* core and server scale out of not only alignment but many bioinformatics processes. Persona has *negligible* overhead, and does not restrict users to specific storage systems or parallel patterns. The dataflow architecture can support different models of parallelism, while the Python API allows user composable pipelines. AGD provides scalable, high-bandwidth access to data. Both Persona and AGD are also extensible, making it easy to integrate new or existing tools and data schemas.

8 Conclusion

In this paper, we demonstrate that existing state-of-the-art bioinformatics tools can be embedded in a distributed dataflow framework based on Google TensorFlow, yielding a composable bioinformatics pipeline that scales linearly with near-zero overhead. In addition, we propose a new data format for genomic data (AGD) that allows for efficient data partitioning and distribution across clusters.

When using the SNAP algorithm, Persona aligns a peak throughput of 1.353 gigabases per second on 32 servers. It can align a 223 million read dataset in $\sim$16.7 seconds. As far as we are aware, this represents the fastest genomic sequence alignment system to date.

When scaled up, alignment can be very cost-efficient, at only 6.07¢ per alignment, showing that bioinformatics computing can be both fast and cost effective. Costs for sequencing, at least in the near future, will be dominated by the cost of consumables and data storage.

Persona and AGD are under active development, with work ongoing to integrate comprehensive data filtering and variant calling. The goal of Persona is to bring the many disparate bioinformatics tools and algorithms into a single, high-performance, yet easy-to-use system that will meet the needs of both small-scale research and large-scale personalized medicine. We look forward to working with the systems and bioinformatics communities to achieve this end.

Acknowledgements

We thank the anonymous reviewers for their constructive feedback and our shepherd, Fred Douglis for his suggestions. This work was supported in part by the NanoTera *YINS* project, Microsoft-EPFL Joint Research Center, and a grant from VMware.

References

[1] 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. J., HARP, A., IRVING, G., ISARD, M., JIA, Y., JZEFOWICZ, R., KAISER, L., KUDLUR, M., LEVENBERG, J., MANÉ, D., MONGA, R., MOORE, S., MURRAY, D. G., OLAH, C., SCHUSTER, M., SHLENS, J., STEINER, B., SUTSKEVER, I., TALWAR, K., TUCKER, P. A., VANHOUCKE, V., VASUDEVAN, V., VIÉGAS, F. B., VINYALS, O., WARDEN, P., WATTENBERG, M., WICKE, M., YU, Y., AND ZHENG, X. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR abs/1603.04467* (2016).

[2] ABUÍN, J. M., PICHEL, J. C., PENA, T. F., AND AMIGO, J. Sparkbwa: speeding up the alignment of high-throughput dna sequencing data. *PloS one 11*, 5 (2016), e0155461.

[3] 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.

[4] AMAZON, I. Amazon glacier pricing. `https://aws.amazon.com/glacier/pricing/`. Accessed: 10-16-2016.

[5] BLUMOFE, R. D., AND LEISERSON, C. E. Scheduling Multithreaded Computations by Work Stealing. *J. ACM 46*, 5 (1999), 720–748.

[6] BROWN, S. M. *Next-Generation DNA Sequencing Informatics*. Cold Spring Harbor Laboratory Press Cold Spring Harbo, 2013.

[7] CHE, Y.-T., CONG, J., LEI, J., LI, S., PETO, M., SPELLMAN, P., WEI, P., AND ZHOU, P. CS-BWAMEM: A Fast and Scalable Read Aligner at the Cloud Scale for Whole Genome Sequencing (Poster). *HiTSeq* (2015).

[8] COCK, P. J., FIELDS, C. J., GOTO, N., HEUER, M. L., AND RICE, P. M. The sanger fastq file format for sequences with quality scores, and the solexa/illumina fastq variants. *Nucleic acids research 38*, 6 (2010), 1767–1771.

[9] DANECEK, P., AUTON, A., ABECASIS, G., ALBERS, C. A., BANKS, E., DEPRISTO, M. A., HANDSAKER, R. E., LUNTER, G., MARTH, G. T., SHERRY, S. T., ET AL. The variant call format and vcftools. *Bioinformatics 27*, 15 (2011), 2156–2158.

[10] DEAN, J., AND BARROSO, L. A. The tail at scale. *Commun. ACM 56*, 2 (2013), 74–80.

[11] DIMAKIS, A. G., GODFREY, B., WU, Y., WAINWRIGHT, M. J., AND RAMCHANDRAN, K. Network coding for distributed storage systems. *IEEE Trans. Information Theory 56*, 9 (2010), 4539–4551.

[12] EBERLE, M. A., FRITZILAS, E., KRUSCHE, P., KALLBERG, M., MOORE, B. L., BEKRITSKY, M. A., IQBAL, Z., CHUANG, H.-Y., HUMPHRAY, S. J., HALPERN, A. L., KRUGLYAK, S., MARGULIES, E. H., MCVEAN, G., AND BENTLEY, D. R. A reference dataset of 5.4 million human variants validated by genetic inheritance from sequencing a three-generation 17-member pedigree. *bioRxiv* (2016).

[13] EPFL VSLC-DCSL. Persona - A High-Performance Bioinformatics Framework. `https://github.com/epfl-vlsc/persona`.

[14] FAUST, G. G., AND HALL, I. M. Samblaster: fast duplicate marking and structural variant read extraction. *Bioinformatics* (2014), btu314.

[15] FRITZ, M. H.-Y., LEINONEN, R., COCHRANE, G., AND BIRNEY, E. Efficient storage of high throughput dna sequencing data using reference-based compression. *Genome research 21*, 5 (2011), 734–740.

[16] GARRISON, E., AND MARTH, G. Haplotype-based variant detection from short-read sequencing. *arXiv preprint arXiv:1207.3907* (2012).

[17] GENOMICS ENGLAND (NHS). The 100,000 Genome Project. `https://www.genomicsengland.co.uk`, 2016.

[18] GEORGANAS, E., BULUÇ, A., CHAPMAN, J., OLIKER, L., ROKHSAR, D., AND YELICK, K. A. merAligner: A Fully Parallel Sequence Aligner. In *Proceedings of the 29th IEEE International Symposium on Parallel and Distributed Processing (IPDPS)* (2015), pp. 561–570.

[19] GONZLEZ-DOMNGUEZ, J., HUNDT, C., AND SCHMIDT, B. parsra: A framework for the parallel execution of short read aligners on compute clusters. *Journal of Computational Science* (2017), –.

[20] GUO, S., AND PHAN, V. A distributed framework for aligning short reads to genomes. *BMC Bioinformatics 15*, S-10 (2014), P22.

[21] HAMILTON, J. Overall data center costs. Accessed: 08-13-2016.

[22] HEALTH, I., AND SCIENCES, L. Genomicsdb. `https://github.com/Intel-HLS/GenomicsDB/wiki`. Accessed: 05-05-2017.

[23] HG19 Human Genome Download. `http://hgdownload.cse.ucsc.edu/goldenPath/hg19/bigZips/`. Accessed: 06-20-2016.

[24] HPC LAB – OSU. Parallel Sequence Mapping Tool. `http://bmi.osu.edu/hpc/software/pmap/pmap.html`, 2016.

[25] ILLUMINA, INC. Illumina NovaSeq. `https://www.illumina.com/systems/sequencing-platforms/novaseq.html`, 2017.

[26] INC., G. Broad institute gatk on google genomics. https://cloud.google.com/genomics/gatk. Accessed: 08-13-2016.

[27] INSTITUTE, B. Picard. https://broadinstitute.github.io/picard/. Accessed: 08-10-2016.

[28] JOURDREN, L., BERNARD, M., DILLIES, M.-A., AND LE CROM, S. Eoulsan: a cloud computing-based framework facilitating high throughput sequencing analyses. *Bioinformatics 28*, 11 (2012), 1542.

[29] LANGMEAD, B., TRAPNELL, C., POP, M., AND SALZBERG, S. L. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology 10*, 3 (2009), 1–10.

[30] LI, H., AND DURBIN, R. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics 25*, 14 (2009), 1754–1760.

[31] LI, H., HANDSAKER, B., WYSOKER, A., FENNELL, T., RUAN, J., HOMER, N., MARTH, G., ABECASIS, G., DURBIN, R., ET AL. The Sequence Alignment/map Format and SAMtools. *Bioinformatics 25*, 16 (2009), 2078–2079.

[32] LI, R., YU, C., LI, Y., LAM, T. W., YIU, S.-M., KRISTIANSEN, K., AND WANG, J. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics 25*, 15 (2009), 1966–1967.

[33] MASSIE, M., NOTHAFT, F., HARTL, C., KOZANITIS, C., SCHUMACHER, A., JOSEPH, A. D., AND PATTERSON, D. A. Adam: Genomics formats and processing patterns for cloud scale computing. *University of California, Berkeley Technical Report, No. UCB/EECS-2013 207* (2013).

[34] MCKENNA, A., HANNA, M., BANKS, E., SIVACHENKO, A., CIBULSKIS, K., KERNYTSKY, A., GARIMELLA, K., ALTSHULER, D., GABRIEL, S., DALY, M., ET AL. The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Research 20*, 9 (2010), 1297–1303.

[35] MICROSOFT. Microsoft genomics. https://enterprise.microsoft.com/en-us/industries/health/genomics/. Accessed: 08-13-2016.

[36] MORETTI, C., THRASHER, A., YU, L., OLSON, M., EMRICH, S. J., AND THAIN, D. A Framework for Scalable Genome Assembly on Clusters, Clouds, and Grids. *IEEE Trans. Parallel Distrib. Syst. 23*, 12 (2012), 2189–2197.

[37] NOVOCRAFT TECHNOLOGIES SDN BHD. NovoAlign. http://www.novocraft.com/products/novoalign/, 2016.

[38] PABINGER, S., DANDER, A., FISCHER, M., SNAJDER, R., SPERK, M., EFREMOVA, M., KRABICHLER, B., SPEICHER, M. R., ZSCHOCKE, J., AND TRAJANOSKI, Z. A survey of tools for variant analysis of next-generation genome sequencing data. *Briefings in Bioinformatics 15*, 2 (2013), 256.

[39] PAPADOPOULOS, S., DATTA, K., MADDEN, S., AND MATTSON, T. G. The TileDB Array Data Storage Manager. *PVLDB 10*, 4 (2016), 349–360.

[40] PELLICER, S., CHEN, G., CHAN, K. C., AND PAN, Y. Distributed sequence alignment applications for the public computing architecture. *IEEE transactions on nanobioscience 7*, 1 (2008), 35–43.

[41] REINDERS, J. *VTune performance analyzer essentials*. Intel Press, 2005.

[42] SCHATZ, M. C. CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics 25*, 11 (2009), 1363–1369.

[43] SMITH, T. F., AND WATERMAN, M. S. Identification of common molecular subsequences. *Journal of molecular biology 147*, 1 (1981), 195–197.

[44] THE HDF GROUP. Hierarchical data format version 5. http://www.hdfgroup.org/HDF5, 2000-2010.

[45] TOLHUIS, B., LUNENBERG, J., AND KARTEN, H. Ultra-fast, accurate and cost-effective ngs read alignment with significant storage footprint reduction. http://www.genalice.com/wp-content/uploads/2013/07/GENALICE-poster-HiTSeq-2013.pdf. Accessed: 08-13-2016.

[46] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Symposium on Operating System Design and Implementation (OSDI)* (2006), pp. 307–320.

[47] ZAHARIA, M., BOLOSKY, W. J., CURTIS, K., FOX, A., PATTERSON, D. A., SHENKER, S., STOICA, I., KARP, R. M., AND SITTLER, T. Faster and More Accurate Sequence Alignment with SNAP. *CoRR abs/1111.5572* (2011).

[48] ZHANG, J., LIN, H., BALAJI, P., AND FENG, W. C. Optimizing burrows-wheeler transform-based sequence alignment on multicore architectures. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing* (2013), pp. 377–384.

SPIN: Seamless Operating System Integration of Peer-to-Peer DMA Between SSDs and GPUs

Shai Bergman
Technion

Tanya Brokhman
Technion

Tzachi Cohen

Mark Silberstein
Technion

Abstract

Recent GPUs enable Peer-to-Peer Direct Memory Access (P2P) from fast peripheral devices like NVMe SSDs to exclude the CPU from the data path between them for efficiency. Unfortunately, using P2P to access *files* is challenging because of the subtleties of low-level non-standard interfaces, which bypass the OS file I/O layers and may hurt system performance.

SPIN integrates P2P into the standard OS file I/O stack, dynamically activating P2P where appropriate, transparently to the user. It combines P2P with page cache accesses, re-enables read-ahead for sequential reads, all while maintaining standard POSIX FS consistency, portability across GPUs and SSDs, and compatibility with virtual block devices such as software RAID.

We evaluate SPIN on NVIDIA and AMD GPUs using standard file I/O benchmarks, application traces and end-to-end experiments. SPIN achieves significant performance speedups across a wide range of workloads, exceeding P2P throughput by up to an order of magnitude. It also boosts the performance of an aerial imagery rendering application by 2.6× by dynamically adapting to its input-dependent file access pattern, and enables 3.3× higher throughput for a GPU-accelerated log server.

1 Introduction

GPU-accelerated applications often require fast data transfers between the GPU and storage devices. They combine high I/O demands with heavy computations amenable to GPU acceleration. Thus, application performance is bounded by the throughput of transfers between the disk and the GPU. As high-speed *NVMe SSDs* with multi-GB/s I/O rates are becoming commodity, we expect an increasing number of I/O-intensive applications to benefit from GPU acceleration. In fact, recent AMD Solid State GPUs (SSG) [1] target such I/O intensive workloads by hosting NVMe SSDs on a GPU card.

In order to realize the potential of high speed I/O devices in GPU workloads, all recent discrete GPUs enable *peer-to-peer direct memory access (*P2P*)* to GPU memory from PCIe-attached peripherals [2, 3]. P2P eliminates redundant copies in CPU memory when transferring data between the devices. Without P2P, copying file contents into a GPU buffer requires reading it first into an intermediate CPU buffer, which is then transferred to the GPU. P2P allows direct transfers into GPU memory, improving performance and power efficiency, as has been shown in several prior works [4–8].

Unfortunately, P2P poses significant programming challenges. First, the usage of P2P requires intimate knowledge of low-level hardware constraints. For example, P2P cannot access files at misaligned file offsets [9], and may be slow or unusable across devices in different NUMA nodes [10].

More crucially, P2P actually hurts system performance for a range of popular file access patterns. Figure 1 shows one such example. For short sequential reads P2P is dramatically slower than CPU-mediated I/O. It performs faster only for reads larger than 512KB. In this scenario, CPU-mediated I/O reaps the benefits of the OS read-ahead mechanism, which P2P bypasses.

Finally, the use of P2P in hybrid CPU-GPU producer-consumer workloads is prone to subtle consistency bugs. Consider, for example, a log processing application like fail2Ban [11], accelerated by leveraging GPUs. Using P2P to read recently updated files might result in an inconsistent read if the contents have not yet reached the disk. Furthermore, because P2P is not integrated with the page cache, users would not benefit from the extensive OS efforts to cache file contents.

We conclude that P2P *between SSDs and GPUs is too low-level a mechanism to be exposed directly to developers*. Existing frameworks [4–8] provide non-standard, custom APIs for performing P2P, but rely on the programmer to work around its limitations and to choose the best-performing transfer mechanism for a given ap-

plication scenario. Instead, the OS should hide the subtleties of direct access to storage, exploit existing file I/O optimization mechanisms like read-ahead and page cache, while *dynamically* and *transparently* steering the data path to P2P.

SPIN is a system that achieves these goals by integrating P2P into the file I/O layer in the OS. The programmer uses *standard* `pread` and `pwrite` calls to transfer the file contents to and from the GPU memory, while SPIN seamlessly activates P2P when necessary. Unlike previous works on P2P [4–8] which target GPU-only workloads with large sequential reads, SPIN addresses a broad range of application scenarios with diverse file access patterns and cooperative CPU-GPU processing.

SPIN addresses three key challenges: integration of P2P with the page cache, read-ahead for GPU reads, and invocation of P2P via a direct disk I/O interface.

Combining page cache and P2P. If a GPU read request can be partially served from the CPU page cache, naively reading all the cached data from memory and the rest via P2P might be slower by up to 16× vs. serving the whole request via P2P. We construct a greedy heuristic that solves the underlying scheduling problem for every access, and produces the interleaving schedule that achieves about 98% of the optimal performance (§4.3.1).

GPU read-ahead. Our read-ahead mechanism uses CPU page cache pages to store the contents of prefetched data for GPU reads. However, SPIN prevents page cache pollution by maintaining a separate GPU read-ahead eviction policy that restricts the space used for prefetched contents (§4.3.2).

Direct disk I/O for P2P. Using direct disk I/O interface to invoke P2P SSD-GPU transfers is advantageous because of its tight integration with the file I/O stack, including page cache consistency handling and file offset-to-logical block address mapping. However, direct I/O calls cannot be used with GPU resident pages. We devise a lightweight *address tunneling* mechanism to overcome this problem (§5.1).

We implement and systematically evaluate SPIN in Linux by running standard file system benchmarks, application traces and full applications. We use NVIDIA K40 and AMD R9 Fury GPUs with two Intel P3700 SSDs, both separately and in a software RAID. SPIN tracks or exceeds the performance of the best transfer mechanism for the respective access pattern, with pronounced benefits over P2P for sequential accesses and accesses to cached files. For example, it achieves 10.1GB/s when reading a file from the page cache – 3.8× higher than 2.65 GB/s of P2P in the same configuration (within 5% of the maximum SSD bandwidth). For partially cached files, SPIN is faster than *either* CPU-mediated I/O or P2P in isolation, e.g., by 2× and 20% respectively for 50% cache hits.

SPIN is compatible with virtual block devices such as software RAID, in contrast to the published P2P implementations. SPIN achieves up to 5.2GB/s of file streaming performance from two SSDs in RAID-0 managed by Linux software RAID [12] – the fastest P2P result reported to date, to the best of our knowledge. For comparison, AMD SSG [1] GPUs with the SSD drives on a GPU card [13] reportedly achieve 4GB/s and require custom API and special-purpose hardware.

In real application scenarios, we evaluate a GPU-accelerated log server, an aerial imagery viewer [14], and an image collage creator [15]. SPIN achieves significant speedups for all applications, e.g., 3.3× for the log server. A highly optimized implementation of the collage creator is improved by 29% while requiring modification of only 10 LOC.

Our main contributions are as follows:

- Analysis of programmability and performance limitations of P2P.
- Integration of P2P into the OS file I/O stack, including standard file I/O API, page cache with a transfer interleaving scheduler, read-ahead and enabling P2P via direct I/O.
- Thorough evaluation on synthetic and real workloads for both NVIDIA and AMD GPUs, showing significant performance benefits of SPIN over alternatives.

2 Background

This section provides a brief overview of the system architecture we target in our work.

System architecture. We consider a system where the CPU, discrete GPUs, and NVMe SSD are connected via Peripheral Component Interconnect Express (PCIe) bus. The PCIe switch enables fast *peer-to-peer direct memory access* (P2P) between the GPU and the SSD. P2P allows the SSD to transfer data directly to/from GPU memory, bypassing the CPU.

Mapping GPU memory into process address space. GPUs expose a portion of GPU memory on the PCIe bus (device's BAR) accessible to the CPU. To allow access to this memory from a user mode application NVIDIA's gdrcopy and AMD's OpenCL extensions provide the tools to map it into the process address space.

Direct disk I/O. Direct disk I/O (O_DIRECT) allows file system operations to bypass kernel caches and interact directly with the storage device.

3 Motivation

Prior works [4–8] show that P2P between SSDs and GPUs substantially boosts system performance for popular GPU benchmarks. These applications exhibit stream-

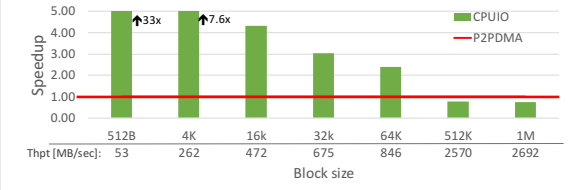


Figure 1: The speedup of CPU-mediated I/O over P2P for sequential reads.

ing access patterns, sequentially reading files in large chunks. Our measurements in this section, however, show that P2P is actually *slower* than CPU-mediated I/O for access patterns and application scenarios that have not been considered previously. We then highlight the key challenges that P2P poses to programmers, motivating its integration into the OS file I/O stack.

3.1 P2P inefficiencies

Short sequential reads. We compare the performance of P2P and CPU-mediated I/O for reading file contents into NVIDIA GPU (AMD GPUs are similar). We run the standard *TIOtest* [16] benchmark only modifying it to transfer data to GPU buffers. The CPU-mediated I/O version issues `pread()` into a CPU buffer followed by `cudaMemcpy()` to transfer the buffer to the GPU. For P2P we use our own implementation described in detail in Section 5.1. For the hardware setup see Section 6.

Figure 1 shows the relative throughput of sequential accesses to a 100MB file. P2P is more than an order of magnitude slower that CPU-mediated I/O for very short reads, and about 3× slower for larger 32KB reads. This is a common access pattern, found, e.g., in grep utility. P2P attains speedups only for reads of 512KB and above.

This performance gap is due to the read-ahead mechanism which transparently optimizes CPU-mediated I/O, and which P2P bypasses entirely. The OS asynchronously prefetches the file into the page cache, overlapping the reads from the disk with CPU-GPU data transfers. The prefetcher gradually increases the size of the prefetch data requests up to 512KB (by default), achieving much higher effective bandwidth to SSD than P2P, which performs short reads.

Complex workloads. P2P is significantly slower than CPU-mediated I/O if the file contents are cached in the page cache, as is often the case for complex software systems with multiple cooperating applications. However, since the page cache contents change dynamically depending on the workload, a programmer is left without a single best choice of file transfer mechanism. For example, consider a central log server that receives logs from other machines over the network and stores them locally. A log scanner invoked as another application might analyze the logs later to detect suspicious events. Using P2P for such a streaming workload might seem as a viable choice. However, if the scanner is invoked immediately after the files are updated, the contents might still be in the page cache, thus using P2P would reduce system throughput, as we also show in our experiments in Section 6.

3.2 P2P programming challenges

P2P is a low-level mechanism, exposed directly to the programmer. Besides the performance issues discussed earlier, it introduces a number of challenges to programmers.

Non-standard API. There is no standard OS API for accessing files via P2P. All the existing frameworks deviate from the standard file API, e.g., `send()/recv()` streaming-like calls in Gullfoss [5] and NVMMU's `move()` [4]. Custom APIs require programmers to explicitly select the file transfer mechanism, a choice that is not trivial in many cases, as we explain earlier.

Data inconsistency. Updates written to a file via regular FS API will be stored in the page cache first, and might remain invisible to the P2P unless the file contents are written back to the disk.

Unsupported misaligned accesses. P2P requires both the source and destination to be aligned according to device-specific rules (p.91, [9]). Specifically, an SSD data offset and destination address must be aligned on the minimum transfer size supported by the device (512 bytes on Intel SSDs), otherwise the I/O request fails.

In summary, as GPUs find their ways to accelerating complex data-processing systems, such as Apache Spark [17], the simplicity, portability, and transparent optimizations offered by OS file I/O interfaces make such interfaces essential for building efficient and maintainable GPU-accelerated systems. These observations guide us in our goal to integrate P2P mechanism into the OS file system layer as we discuss next.

4 Design

Design goals. SPIN aims to integrate P2P into the OS file I/O layer. It uses P2P as a low-level mechanism for optimizing file I/O where applicable. We focus on the following design goals:

- **CPU-GPU workloads:** efficiently handle complex scenarios with opportunistic data reuse, where applications may share files, e.g., in producer-consumer interaction. SPIN should provide standard POSIX file consistency guarantees regardless of the transfer mechanism used.

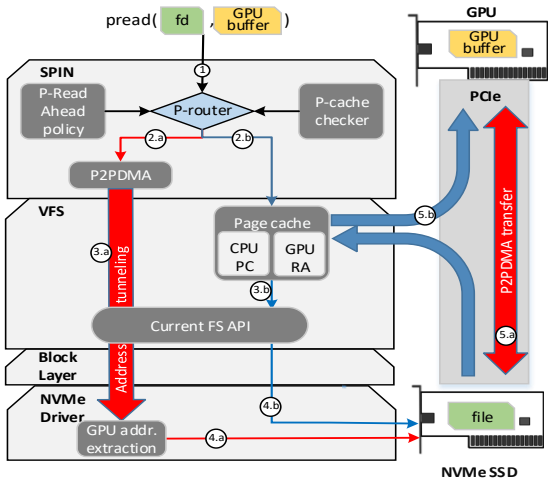


Figure 2: SPIN high-level design and control flow of `pread ()`, as explained in Section 4.2

- **Various access patterns:** enable high performance across random/sequential access patterns and an unrestricted range of request sizes, from as little as a few bytes.
- **Standard File API:** support standard I/O calls like `pwrite ()` and `pread ()` for portability.
- **Compatibility:** be compatible with virtual block devices such as LVM and software RAIDs, as well as with different GPUs and SSDs.

4.1 Design considerations

Page cache is the cornerstone of file I/O in CPU systems, but its integration with P2P raises a number of questions.

Page cache in GPU memory? One way to combine caching with P2P is to partition the page cache between the CPU and GPU memories, and use each to cache file accesses from the respective device. In fact, GPUfs demonstrated the benefits of hosting a page cache for GPU tasks in GPU memory [15, 18]. Unfortunately, modern GPUs still lack critical features to enable OS-controlled GPU-resident page cache. In particular, they do not support anonymous memory that does not belong to any CPU process, neither do they provide the means for the OS to manage GPU memory mappings. As a result, GPUfs, for example, maintains a *per-application* page cache, which disappears when an application terminates. Workarounds, such as running a daemon process in user space that *owns* the GPU page cache, are insecure because they expose the whole page cache to all running GPU tasks. We conclude that *maintaining page cache in GPU memory is currently not practical.*

Reusing file contents from the CPU page cache. P2P transfers bypass the CPU page cache. But if the content is already in the cache, using P2P would be slower than reading the data from the page cache. However, if only part of the request can be served from the cache, the best way to combine P2P and cache accesses depends on the distribution of the pages in the cache. For example, if only every second page in a 8MB-large read request is cached, reading from the page cache is 16× slower than a single P2P of the whole requested buffer. We address the problem of optimal interleaving in Section 4.3.1.

Read-ahead integration. A read-ahead mechanism is essential for fast sequential accesses (see § 3), but the best way to integrate it with P2P is not obvious. Technically, the prefetcher never runs because P2P bypasses the heuristic which identifies a sequential access pattern and triggers the read-ahead mechanism. However if we re-enable the prefetcher, where will it store the prefetched contents? One of the benefits of P2P is that it does not pollute the CPU page cache with the data used only by the GPU. But without the page cache on the GPU, the read-ahead mechanism would have to store the prefetched data in the CPU page cache, losing this advantage. We discuss the prefetcher in Section 4.3.2.

Portability across GPU software. GPU vendors expose different APIs for GPU management and data transfers to and from GPU memory, none of which are available for use from kernel space. As a result, providing a generic OS service which is agnostic to the GPU type and its software stack is challenging.

4.2 Overview

Figure 2 shows the main design components. SPIN is positioned on top of the Virtual File System (VFS) layer. We illustrate the interaction of the SPIN components on the example of `pread ()`. The user allocates the destination buffer in GPU memory and passes the pointer to the buffer to `pread`. To make GPU memory buffers accessible to I/O calls, the user *maps the buffers into the CPU process address space* using existing GPU vendor-specific tools (§ 5). We note that using CPU-mapped GPU buffers in I/O calls is possible without SPIN, however P2P is not invoked.

The SPIN core is implemented in *P-router*. P-router inspects every I/O request (①) in the Figure) and detects the requests that operate on GPU memory buffers and are amenable to P2P. P-router invokes the *P-readahead* mechanism, which identifies sequential access pattern and prefetches file contents into a *GPU read-ahead partition* (GPU RA in the Figure) of the *CPU* page cache, as described in § 4.3.2). It also checks with *P-cache* whether the request can be served from the page cache, and creates an I/O schedule to interleave P2P and page cache accesses, as discussed in § 4.3.1. Finally, it generates VFS I/O requests that are served by a combination of the page

cache (2.b) and P2P (2.a). To invoke P2P via direct disk I/O interface, P-router employs an *address tunneling* mechanism (3.a) described in § 5.1.

4.3 Integration with page cache

We deal with three aspects: interleaving page cache reads with P2P, integration with read-ahead, and data consistency.

4.3.1 Combining page cache with P2P

Optimal scheduling of page cache transfers. P-cache retrieves the CPU page cache residence map for a given read access. If the entire requested region is cached, the request is served from the page cache. However, if the cache contains only part of the requested data, the system combines both P2P and page cache transfers, by breaking the original request into sub-requests each served via its own method.

Finding the best interleaving of P2P and page cache accesses is a challenge. On the one hand, reading from the page cache is faster than reading from the SSD. On the other hand, interleaving P2P and page cache reads at a fine granularity results in poor performance, because short I/O requests to the SSD are less efficient than larger ones, and because of the P2P invocation overhead. Thus, SPIN needs to determine the best interleaving schedule for each I/O request.

The following example illustrates the problem. Consider a request of 20KB (5 pages) with its second, and fourth pages in the page cache. Then, there are 3 possible schedules: three P2P transfers of 4KB and two 4KB transfers from page cache, a single P2P of 20KB of the whole range, and a combination of P2P and page cache transfers for the second and the fourth page, resulting in two P2P transfers of 4KB and 12KB. The choice of the best schedule depends on the actual P2P throughput for each transfer size, as well as on the throughput of the page cache reads. The scheduling decision for different pages are *not* independent, however, because SSD transfer time is a non-linear function of the request size for smaller reads [19].

To summarize, the scheduling problem at hand is as follows: for a given I/O request, find all the constituent continuous ranges of pages which can be served from the page cache. For every such a range, decide whether to transfer it from the page cache or via P2P, effectively merging it with the two flanking segments into a single P2P transfer, such that the total transfer time of the whole request is minimized.

Greedy heuristic. This problem can be solved exactly in polynomial time via dynamic programming, however this is too slow since the solution has to be found for every I/O request. Instead, we simplify the problem to apply a simple greedy heuristic as follows.

We start by assuming that the P2P transfer time, $T_{p2p}(s)$, is a piece-wise linear function of the transfer size s of the form given in Eq 1. Intuitively, for requests smaller than S_{cutoff}, the device bandwidth is not saturated, thus the transfer time is constant and capped by the device's invocation overhead C_{p2p}. For requests larger than S_{cutoff}, the device operates at maximum bandwidth BW_{p2p}. These assumptions are consistent with the architectural model of modern SSDs [19]. Page cache transfers, in turn, always achieve maximum bandwidth thus the transfer time for size s is $T_{pc}(s) = \frac{s}{BW_{pc}}$.

$$T_{p2p}(s) = \begin{cases} C_{p2p} & \text{if s} < S_{cutoff} \\ C_{p2p} + \frac{s - S_{cutoff}}{BW_{p2p}} & \text{if s} \geq S_{cutoff} \end{cases} \quad (1)$$

The greedy heuristic works as follows. For each three consecutive data ranges a, b, c, where b is in the page cache, if $|a| + |b| < S_{cutoff}$, always choose P2P for b (where $|x|$ is the size of x). Otherwise, choose P2P for b if $T_{p2p}(|a| + |b| + |c|) < T_{p2p}(|a|) + T_{pc}(|b|) + T_{p2p}(|c|)$. In other words, P2P for b is preferable if the benefits of reading b from the page cache are smaller than the overhead of transferring c in a separate P2P transaction.

Parameter fitting. We experimentally measure the transfer times for different request sizes for Intel P3700 SSD, and fit the parameters of the transfer time function in Eq 1 using regression. The function fits very well, with the coefficient of determination of over 0.99. We find $S_{cutoff} = 512$KB and $C_{p2p} = 584\mu sec$, which corresponds to the time for transferring 249 pages from the page cache. Thus, for two consecutive data ranges b, c where b is in the page cache and c is not, b will be always transferred via P2P if $|b| < 249$ pages.

Evaluation. We build a simulator which quickly computes the transfer cost of an I/O request, given transfer schedule, using the transfer times measured on real hardware. We validate the simulator experimentally on 5,000 I/O requests, and find that its error is 1.6% on average.

We use the simulator to evaluate the quality of the greedy heuristic, by comparing its results with the optimal transfer schedules obtained by the exact algorithm. We evaluate the schedules on 200,000 random vectors, each representing an 8MB data transfer having different page cache residency patterns. We find that the transfer time of the greedy schedules is within 98.9% of the optimal schedule on average.

Generalization to other SSDs. We believe that our heuristic reflects the general SSD performance trends and can be used with other SSDs. Specifically, architectural properties of SSDs, such as multi-channel/multi-way, enable a high degree of parallelism for relatively large re-

quests. These requests are often striped across domains and exploit the internal parallelism SSDs offer [19, 20]. Therefore, our model which predicts higher performance for larger requests is consistent with these properties. We provide a calibration tool to perform the measurements and regression to automatically adjust S_{cutoff} and C_{p2p}.

4.3.2 Read-ahead for GPU accesses

The OS read-ahead is not activated for accesses via P2P, therefore we introduce *P-readahead*. It stores the prefetched data in a special partition in the CPU page cache as we explain next.

GPU read-ahead cache. To avoid cache pollution by the contents prefetched as part of the read-ahead, we add a lightweight management mechanism, *GPU read-ahead cache, RA cache*. A page is assigned to the RA cache when it is first used by P-readahead to store the prefetched data. The pages in the RA cache belong to the OS page cache, and are subject to OS page cache management policies. In addition, the RA cache forces eviction of its pages once its total size exceeds a predefined threshold. If a page is later accessed by a CPU program, the page is removed from the RA cache, but remains in the OS page cache. As a result, the pages used exclusively to store the data prefetched for GPU I/O do not pollute the OS page cache.

Read-ahead mechanism. P-readahead watches for sequential access pattern by monitoring the last accessed offset in each file, similarly to the CPU read-ahead heuristic. For sequential accesses, the data is read into the GPU RA cache via CPU VFS calls, effectively engaging the original OS read-ahead mechanism redirected to store data in the GPU RA page cache. As a result, P-readahead respects the standard `fadvise` calls, and does not require new management interfaces. We also modify the default behavior of P-readahead in response to `fadvise` policies, e.g., disabling it for `POSIX_FADV_RANDOM`.

For sequential requests that cannot be served from the page cache and exceed a certain threshold, P-router deactivates P-readahead and switches to P2P. The threshold equals to the maximum size of the OS-configured read-ahead window (512KB by default), which determines the maximum size of SSD requests generated by the read-ahead. Using P2P for requests exceeding the threshold results in larger SSD requests and higher throughput.

4.3.3 Data consistency

Combining file accesses from the page cache with direct accesses to a storage device raises an obvious data consistency problem, since the data in the page cache might not be synchronized with the content on the SSD. Therefore, SPIN detects dirty pages in the range of the P2P transfer, and explicitly performs a write back from the page cache to the SSD.

5 Implementation

Our implementation leverages existing kernel mechanisms to achieve SPIN's design goals. We encapsulate all new functionality in a kernel module SPINDRV, a slightly modified generic NVMe driver, and a lightweight user space library LIBSPIN. Thus, SPIN requires no modifications to the kernel and is readily deployable on existing systems.

libSPIN. is a shim that interposes on standard file I/O calls. The library is loaded via an `LD_PRELOAD` environment variable. Applications that do not load LIBSPIN may share files with those that do.

Interaction with GPUs. SPIN leverages existing tools for mapping GPU memory into the CPU address space. In particular, we use OpenCL's CL_MEM_USE_PERSISTENT_MEM_AMD extension from AMD, and `gdrcopy` module from NVIDIA. Using CPU-mapped GPU memory for I/O enables portability across GPU vendors, interaction with GPUs from kernel space, and independence from GPU software interfaces.

SPINdrv. The driver implements the SPIN design including the page cache and read-ahead as described in § 4. In addition, it introduces a new *address tunneling* mechanism to enable P2P via direct disk I/O which we discuss next.

5.1 P2P via direct disk I/O

Our implementation of P2P takes advantage of the direct disk I/O file interface, adding a special mechanism to enable its use with GPU memory buffers.

Direct disk I/O and P2P pursue the same goals: they allow direct access to storage devices while bypassing the OS page cache. Using direct disk I/O mechanisms for P2P has a number of advantages. First, the file I/O stack performs the standard file offset-to-LBA mapping which is compatible with virtual block layers, e.g., software RAID. Second, the mechanism already implements various optimizations, e.g. uses multiple submission queues and merges/splits block I/O requests. Last, it already handles the data consistency by writing back dirty page cache pages in the range of its I/O request.

Unfortunately, direct disk I/O requires the user buffers to reside in CPU physical memory, and cannot accommodate CPU-mapped GPU buffers. This is because it pins user buffers in memory to perform DMA to/from the storage device, and fails to pin GPU buffers. This problem has no easy solution, as we discuss below (§5.2).

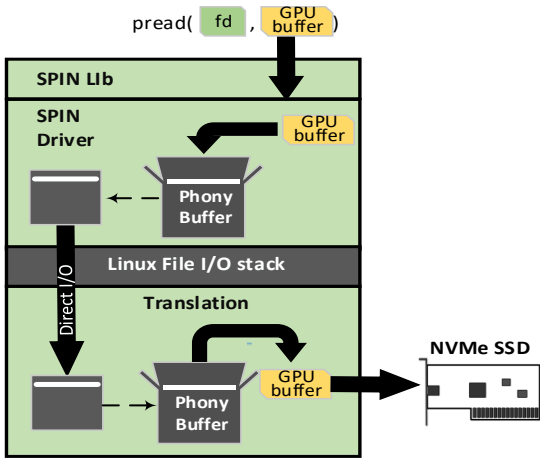


Figure 3: Address tunneling for direct disk I/O with GPU buffers.

Address tunneling. To overcome this limitation without major modifications to the Linux kernel, we design a simple mechanism that we call *address tunneling*, which delivers the GPU address through unmodified VFS stack and block layers down to the generic NVMe driver.

Figure 3 explains the basic idea. We allocate a special user-space *phony buffer* in the CPU, which is used as an envelope for the GPU buffer address. The phony buffer is then passed to a VFS file I/O call, instead of the original GPU buffer. Therefore, it successfully undergoes all the translation and pinning process while passing through intermediate I/O layers. When the envelope reaches the generic NVMe driver, the driver retrieves the address of the GPU buffer and uses this address to perform P2P.

Security of address tunneling. One potential problem with the tunneling mechanism is that phony buffers are allocated in the user-space memory (otherwise they cannot be passed to VFS calls), hence they are accessible to user-space programs and can be overwritten by an adversary to potentially hold any physical CPU address, thereby enabling DMA attacks. SPIN, therefore, does not store the actual GPU addresses in phony buffers. Instead, it first creates a temporary pseudo-random token associated with the current request, and uses the token as the key to the kernel-space translation table with the actual GPU addresses.

Implementation details. The phony buffer is a user space CPU memory buffer allocated once during the process invocation when LIBSPIN is loaded. The buffer is pinned in memory and registered with the SPINDRV. Its size remains constant (currently 4MB) throughout the execution. Since an I/O request must fit in the phony buffer, the I/O requests larger than 4MB are split into multiple requests. Each memory page of the phony buffer is used to store the address of one page in the GPU buffer, since the block layer may reorder the requests and split them into smaller chunks.

Multiple threads in the same process may use the same phony buffer simultaneously in a lockless manner. One thread consumes only 16 bytes per a GPU address in a 4K page, therefore a single phony buffer may accommodate up to 256 concurrent requests from different threads.

The GPU read-ahead cache is implemented as a linked list that references 512 pages (tunable), located in the OS page cache. The eviction is policy is LRU. Pages accessed by a CPU program are simply removed from the list, and are not evicted from the OS page cache itself.

Interaction with generic NVMe driver. The phony buffer's pages are marked by setting an unused (for user mode) `arch_1` flag in their `page_struct`. This flag is used by the driver to differentiate P2P requests from regular pages and extract the GPU addresses.

Implementation complexity. SPINDRV is implemented in 700 LOC and LIBSPIN just 30 LOC. We modified 10 LOC in the Linux generic NVMe driver to detect phony buffers and extract respective GPU addresses.

5.2 Limitations

Supporting `pwrite()`. Mapping GPU memory into the process's address space is a recent capability that is not yet well supported in current systems. Specifically, CPU reads from that memory mapping are about two-three orders of magnitude slower than CPU writes [21], i.e., about 30MB/s and 70MB/s for NVIDIA and AMD GPUs respectively. Therefore, while reading data from the page cache into the GPU is fast, writing files from the GPU into the page cache – which might be beneficial e.g., for buffering writes in CPU memory – results in severe performance degradation. Therefore, we currently configure SPIN to perform writes from GPU memory only via P2P, while taking care of data consistency.

Changing Linux to natively support GPU buffers. The address tunneling mechanism sidesteps the problem of passing GPU buffers to direct disk I/O, but why not changing the kernel in the first place? Technically, the problem originates in the use of `struct_page` which is not available for I/O re-mapped addresses such as GPU memory buffers. However, this struct is required by the block layer. Attempts have been made to solve the problem in a systematic way [22], yet they require touching over 100 files of kernel code. We therefore choose a more conservative solution.

6 Evaluation

We evaluate SPIN on two hardware systems (Table 1). We disable HyperThreading and configure the frequency

Nvidia Tesla K40c	2 × Intel Xeon E5-2620v2, Intel C602 Chipset, 64GB DDR4, 1 NVMe SSD
AMD Radeon R9 Fury	Intel Core i7-5930K, Intel X99 Chipset, 24GB DDR4, 2 NVMe SSDs

Table 1: Evaluation platforms. Both use one or two Intel P3700 800GB NVMe SSD

ClWrite	Regular `read` into the CPU, followed by a blocking `clEnqueueWriteBuffer / cudaMemcopy` call to the GPU.
ClWrite+D	Same as ClWrite but with bypassing the CPU page cache via O_DIRECT flag.
P2P	SPIN's implementation of P2P that bypasses the page cache.
pread+GPU	`pread` into the GPU memory that is mapped to the process's address space. Unlike SPIN, `pread` ()+GPU always uses the page cache. Not evaluated in prior works.

Table 2: Transfer mechanisms used for evaluation.

governor to high performance to reduce overall system noise. Both machines run Ubuntu 15.04 with and untainted Linux kernel 3.19.0-47 and ext4 on SSD. We use CUDA 7.5 for NVIDIA and OpenCL 2.0 for AMD.

Methodology. We run each experiment 11 times, omit the first result as a warmup, and report the average of the last 10 runs. We explicitly flush the contents of the page cache before each run (unless stated otherwise). We observe the standard deviation below 1% across all the experiments and do not report it in the figures.

Alternative transfer methods. We compare SPIN with several different implementations described in Table 2. We note that the implementation where `pread` () is invoked with the CPU-mapped GPU buffer (last row) has not been evaluated in prior works.

Alternative implementations of P2P**.** Although several prior works reportedly implement P2P between SSDs and GPUs [4–8], we found only the early prototype of Project Donard [8] to be publicly available. However, this prototype is limited and is slower for all request sizes, and particularly for shorter requests, therefore we do not include it in the experiments.

6.1 Threaded IO benchmarks

We use TIOtest [16] for our benchmarks. TIOtest is a standard tool for evaluating file I/O performance in CPU-only systems. It supports multi-threading (each thread accesses its own file), sequential/random access patterns and different I/O request sizes. We modify the original code [1] to read data into GPU buffers using all the five evaluated implementations. For SPIN our changes required modifying 10 LOC for buffer allocation.

[1] `https://wiki.codeaurora.org/xwiki/bin/Linux+Filesystems/Tiobench`

We report the results for the AMD GPU, and discuss the performance of the NVIDIA GPU in the text.

Random Reads. In this experiment each worker thread reads 500 blocks at random offsets from a 50GB thread-private file. Figure 4a shows the results. Note that the drops in the relative throughput on the graph do not imply lower absolute throughput, rather they mean slowdown compared to SPIN in the respective configuration. The results for a single CPU thread are similar and omitted due space limitations.

SPIN performance matches the one of P2P, adding only 1% overhead. For blocks above 1MB the overhead of additional memory copy in CPU memory gets amortized for all the implementations but ClWrite, because of its second extra copy in the temporary CPU buffer.

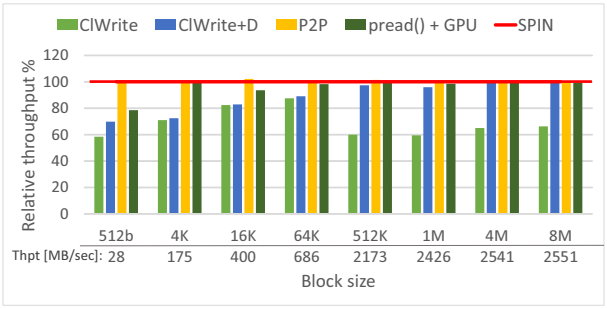
Sequential reads. For sequential reads, each worker thread in TIOtest reads an entire file of 100MB. Figure 4b shows that SPIN tracks the best performing method for the specific block size, switching from page cache to P2P at 512KB as explained in Section 4.3.2. We observe that for blocks smaller than 4K SPIN experiences higher relative overhead of up to 10% because it serves them from the page cache. The overhead is amortized for larger reads, however.

Sequential/random writes. For the sequential writes, each worker thread writes a 100MB file. The `pwrite` +GPU mechanism is dramatically slower than P2P, as we explain in Section 5.2, therefore SPIN always performs aligned writes via P2P. Random writes perform similarly. Due to the lack of space, the figure is omitted.

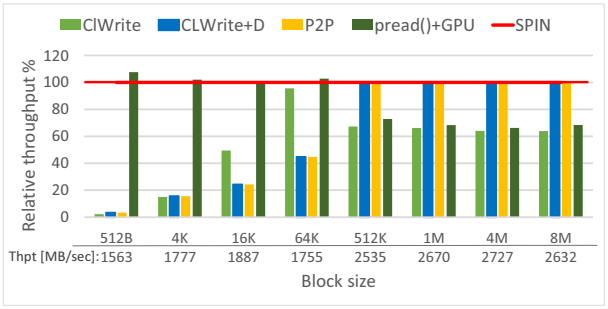
Performance on NVIDIA and AMD GPUs. SPIN achieves 5-10% higher throughput on AMD R9 GPU than on NVIDIA K40C GPU, while the overall behavior is similar. We find that cudaMemcopy might be slower then AMD ClWrite, and the GPU BAR writes for NVIDIA GPUs are slower for some block sizes. These results indicate that SPIN works well with GPUs from different vendors, however the small performance gap we observe requires further investigation.

Software RAID-0 . We use the standard mdadm Linux utility to create a RAID-0 (striping) volume over two NVMe SSDs. In this configuration, the stored data is split between two SSDs according to the configured stripe size (512KB in our configuration), thus performing larger file accesses in parallel.

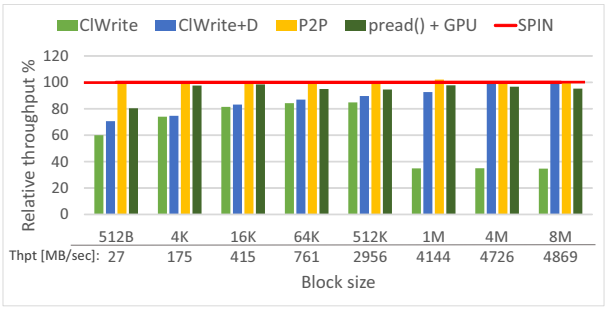
Figure 4c shows the relative throughput of random accesses for which SPIN always uses P2P. RAID-0 outperforms a single SSD only for large reads (above 512KB). This is due to extra overheads of additional processing in the RAID layer which get amortized for larger blocks. For large sequential reads, SPIN achieves a throughput of 5.2GB/s. The higher bandwidth is due to the SSDs performance characteristics.



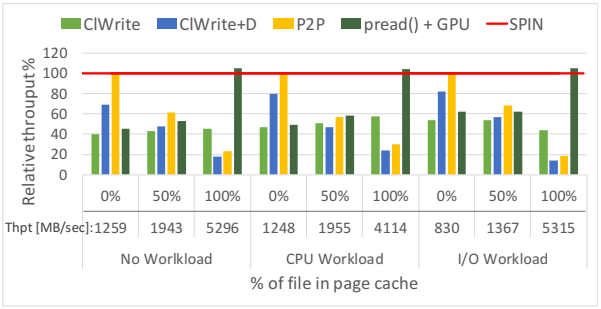
(a) Random reads, 4 threads



(b) Sequential reads, 4 threads



(c) RAID: Random reads, 4 threads



(d) Random 512KB reads, in parallel with CPU / I/O workloads

Figure 4: Threaded IO benchmarks for AMD GPUs.

SPIN	`pread` + GPU	P2P	P2P + RAID	ClWrite	CLWrite+ D+RAID
10.13	10.28	2.65	5.29	5.72	4.69

Table 3: Max read throughput (GB/s). File in page cache.

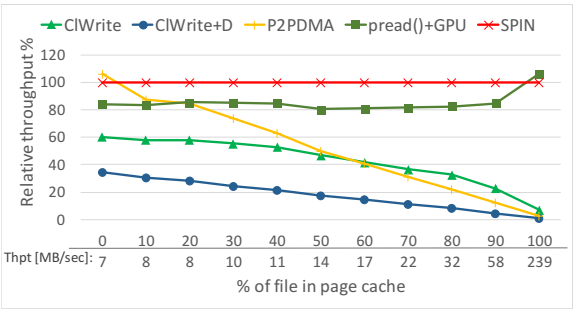


Figure 5: Random access performance for different page cache occupancy. Reading blocks of 512B.

Maximum sequential read throughput. We compare the maximum achievable throughput over different transfer mechanisms. The test performs sequential reads from 4 threads, 8MB per read from a 4GB file, when a file is *prefetched into the page cache*. Table 3 shows the results. SPIN is faster than all the transfer methods that do not use page cache, and faster than ClWrite that does. SPIN's overhead in this scenario is 1.5%.

Effect of the page cache on read throughput. The goal of this experiment is to show potential performance gains for producer-consumer workloads which may utilize both the CPU and GPU while they access a shared file. We prefetch different portions of a 40GB file into the page cache using `vmtouch` [2], and run TIOtest for 512B random reads.

Figure 5 shows the relative throughput, highlighting the differences between transfer methods. Not only does SPIN track the best alternative, it is *faster* than the fastest among them by up to 20%. That is because it combines both page cache and P2P, dynamically choosing between them per request depending on the residence in the page cache (discussed in (§ 4.3.1)). SPIN is slightly slower on the extremes due to the 5% overhead it introduces in this scenario. ClWrite results in low performance due to its constant invocation overhead, whose relative weight grows when most requests are served from the page cache, as we also see in Figure 4b.

SPIN performance under CPU and I/O load. We execute the same experiment as in Figure 5, but now impose heavy load on all the CPUs or SSD in parallel with the benchmark. The benchmark performs 512KB random reads (cutoff size for reading from the page cache), to show the worst-case scenario for SPIN under CPU load.

[2] https://hoytech.com/vmtouch/

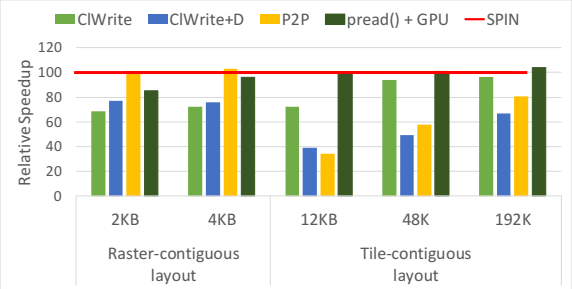


Figure 6: Aerial imagery benchmark throughput relative to SPIN for different file layouts. Higher is better.

We use `stress-ng` [3] benchmarking tool. Figure 4d shows the relative throughput for 0%, 50%, and 100% file residency in the page cache, with and without CPU or SSD load. We observe that SPIN retains its performance advantages regardless of the system load.

6.2 Application benchmarks

Aerial Imagery Rendering. GPUs are commonly used for rendering aerial imagery in *geographic information systems (GIS)*. The datasets used in such systems may grow to hundreds of GBs. Large rasters are split into tiles in order to shorten system response time. The rendering engine reads the tiles from a file depending on the view point, and stitches them together.

In our evaluation we generate I/O traces via a benchmarking tool for web-based rendering engines [23]. We use TrueMarble dataset [24] from standard benchmarks [23], which is a 190GB multi-raster of the Earth, each raster corresponds to a different image resolution.

The actual file access pattern in this application depends on the underlying file layout. There are two layouts: (1) raster-contiguous layout, where the whole raster is stored as a 1D vector in the file and (2) tile-contiguous layout, where each *tile* is a 1D vector and the raster is composed of many 1D tiles. The first layout results in mostly random accesses 2-4KB each, whereas the second involves mostly sequential accesses each from 12KB to 192KB. We emphasize that the rendering applications must be able to accommodate files with both layouts.

To generate the trace we randomly choose the target image resolution and the view region, derive the tiles to render that region and record their respective offsets in the dataset file. We use tiles of sizes ranging from 64x64 pixels up to 1024x1024 pixels. In every trace we emulate rendering of 1000 different regions in full HD.

We generate the traces for different input layouts and compare the throughput of different transfer mechanisms. As Figure 6 shows, *the choice of the transfer mechanism depends on the layout in use*. For the native layout with mostly random access pattern, P2P and SPIN achieve the highest throughput. However, for tiled layout the reads are mostly sequential, and SPIN benefits from the read-ahead achieving up to 2.5× higher throughput than P2P for 12K reads. SPIN eliminates the need to manually perform such low level optimizations, reducing code complexity and development efforts.

[3] https://openbenchmarking.org/test/pts/stress-ng

Configuration		CPU	GPU		
			P2P	`ClWrite()`	SPIN
R-time	Thput	771	594 (0.8×)	1921 (2.5×)	1950 (2.5×)
	CPU util	79.5%	3%	11.8%	10.7%
Offline	Thput	634	2549 (4×)	1822 (2.9×)	2550 (4×)
	CPU util	70.3%	8.5%	12.3%	8.5%

Table 4: Log server throughput (in MB/s), CPU utilization and speedup over the CPU-only version

GPU-accelerated log server. Log servers, such as VMWare VRealize [25], are commonly used in distributed systems for centralized storage and processing of logs from multiple servers. Log processing usually involves string and regular expression matching, which may benefit from acceleration on GPUs [26].

We implement a simple log server which receives log files over the network, stores them locally in files, and scans them for suspicious IPs from the list provided by the user. As is common in log processing systems, e.g., Fail2Ban [11], log analysis is performed in a separate scanner process that reads the specified log file and processes it. Such a modular design is convenient because it enables to easily extend the analysis using several independent backends. Our implementation of the scanner offloads the string matching to a GPU.

We measure the maximum system throughput in two scenarios: (1) real time, in which the scanner is invoked each time the files get updated (using `inotify` interface) (2) offline, in which the scanner is invoked on a specific log file to be processed as a whole. In both configurations, a total of 80GB of data is processed.

We evaluate our GPU implementation with different I/O mechanisms: (1) traditional pread() followed by `ClWrite()` to GPU memory, (2) P2P (3) SPIN. We also implement a CPU-only version that uses Intel's Threading Building Blocks and runs on 6 cores.

Table 4 shows that in the *real time* scenario SPIN achieves the highest throughput among all other I/O methods. Since the system triggers log processing right after it receives log file updates from the network, the new contents have not yet been written back to the disk and reside entirely in the page cache. SPIN, therefore, reads the data from the page cache, relieving I/O contention on the SSD which do occur in P2P configuration. In the steady state, the system throughput is limited by the maximum SSD write throughput, because the net-

work server keeps writing the updates to storage, eventually exhausting the page cache space. In the *offline* scenario the data is not in the page cache, therefore SPIN switches to use P2P.

In this application, complex interactions between multiple processes dynamically create file data reuse opportunities that cannot be known in advance, hence are hard to leverage without the OS support. SPIN re-enables the standard OS ability to handle such opportunistic reuse automatically for file transfers to the GPU.

Image collage. The image collage application [15] creates an image collage by replacing blocks in the input image with "similar" tiny images from a data base (we use [27]). Pre-processed tiny images are stored in a file of size 38GB. We use an open-source implementation that uses GPUfs [18] GPU-side library for accessing files from GPU kernels. GPUfs uses a dedicated worker thread running on the CPU to handle the file transfers into the GPU memory. This application performs mostly random reads 512B each.

The original version of GPUfs first reads the file contents into the host staging area, and then copies the data into GPU memory via `cudaMemcopy`. We remove the staging area in the host, and allocate the staging area in the GPU memory, changing in total 30 LOC.

We measure the SPIN speedup over the unmodified version. For three different input images of 3MB, 12MB and 48MB SPIN is $\times 1.27 \pm 0.02$ faster on average thanks to the use of P2P for short random reads.

7 Related work

System support for P2P. There have been several works which enable P2P between NVMe SSDs and GPUs, but SPIN is the first to integrate P2P with the OS file I/O, dealing with page cache, read-ahead, data consistency, and compatibility with virtual block devices.

GPUDrive [6] is a system for processing streaming I/O-intensive GPU workloads based on an all-flash storage array connected to the GPU.

NVMMU [4] introduces a special programming model and runtime for P2P with GPUs. NVMUU shows that P2P achieves high performance with standard GPU compute benchmarks modified to read input data from files. Unlike SPIN, however, it requires a custom interface for P2P, does not address the page cache integration issues, and focuses only on GPU-only applications with large sequential reads. In fact, it shows that P2P is slow for small I/O requests but does not address this problem.

Project Donard [8] was among the first to support P2P via a low level driver interface. Among its many limitations, it runs only with root privileges due to direct access to NVMe DMA, and suffers from performance issues. Gullfoss [5] software framework for P2P shares many conceptual similarities with NVMMU, and hence many of its limitations. Morpheus [7] enables P2P to GPUs from SSDs, but does not address the challenges of integrating P2P into standard file I/O, focusing primarily on low level P2P functionality.

GDRcopy [21] uses CPU-mapped regions of GPU memory for efficient data transfers to GPUs. SPIN leverages the same functionality.

P2P technologies. Recent GPUs offer support for P2P, including GPUDirectRDMA [28] from NVIDIA and DirectGMA [3] from AMD. These technologies provide generic support for direct access to GPU memory from PCIe devices, but they do not integrate it into higher level services like file I/O.

System abstractions for GPUs. GPUfs and GPUnet [10, 18, 29] provide file access and networking directly to GPU programs. The current work is complementary as it simplifies the use of P2P for *CPU* programs.

8 Conclusions

SPIN focuses on the fundamental problem of providing generic OS abstractions in heterogeneous systems, extending the traditional I/O mechanisms to systematically deal with direct I/O into the GPU. We show the importance of tighter integration of P2P with the file I/O stack, expose the challenges associated with the use of P2P together with the page cache and read-ahead, and devise a practical solution which outperforms the state-of-the-art in a range of realistic scenarios.

Current hardware trends are toward systems with multiple accelerators [30, 31], which will dramatically increase system heterogeneity and complicate software development. OS support for such increasingly heterogeneous systems must extend beyond low-level APIs, and provide the convenience of high level OS abstractions to achieve their performance potential. SPIN is a step in this direction.

SPIN is available at https://github.com/acsl-technion/spin

Acknowledgements

Mark Silberstein is supported by the Israel Science Foundation (grant No. 1138/14), and the Israeli Ministry of Economics via HiPer consortium.

References

[1] "AMD Radeon Pro SSG Set to Transform Workstation PC Architecture, and to Shatter Real-Time Visual Computing Barriers."

http://www.amd.com/en-us/press-releases/Pages/amd-radeon-pro-2016jul25.aspx, 2016.

[2] "GPUDirect RDMA." http://docs.nvidia.com/cuda/gpudirect-rdma/index.html, 2015.

[3] "Tech Brief: AMD FirePro™ SDI - Link and AMD DirectGMA Technology." https://www.amd.com/Documents/SDI-tech-brief.pdf.

[4] J. Zhang, D. Donofrio, J. Shalf, M. T. Kandemir, and M. Jung, "NVMMU: A Non-volatile Memory Management Unit for Heterogeneous GPU-SSD Architectures," in *PACT*, pp. 13–24, IEEE, 2015.

[5] H.-W. Tseng, Y. Liu, M. Gahagan, J. Li, Y. Jin, and S. Swanson, "Gullfoss: Accelerating and Simplifying Data Movement Among Heterogeneous Computing and Storage Resources," *Tech. Rep. CS2015-1015, Department of Computer Science and Engineering, University of California, San Diego technical report*, 2015.

[6] M. Shihab, K. Taht, and M. Jung, "GPUDrive: Reconsidering Storage Accesses for GPU Acceleration," in *Workshop on Architectures and Systems for Big Data*, 2014.

[7] H.-W. Tseng, Q. Zhao, Y. Zhou, M. Gahagan, and S. Swanson, "Morpheus: creating application objects efficiently for heterogeneous computing," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 53–65, IEEE, 2016.

[8] "Project Donard." https://github.com/sbates130272/donard, 2015.

[9] "NVM Express 1.0e." http://www.nvmexpress.org/wp-content/uploads/NVM-Express-1_0e.pdf, 2013.

[10] S. Kim, S. Huh, X. Z. Yige Hu, A. Wated, E. Witchel, and M. Silberstein, "GPUnet: Networking Abstractions for GPU Programs," in *OSDI 14*, pp. 6–8, USENIX, 2014.

[11] "Fail2Ban." www.fail2ban.org/.

[12] "mdadm - manage MD devices aka Linux Software RAID." https://www.kernel.org/pub/linux/utils/raid/mdadm/.

[13] Anandech, "AMD announces Radeon-Pro SSG." http://www.anandtech.com/show/10518/amd-announces-radeon-pro-ssg-fiji-with-m2-ssds-onboard, 2016.

[14] "ArcGIS for Desktop." http://desktop.arcgis.com/en/arcmap.

[15] S. Shahar, S. Bergman, and M. Silberstein, "ActivePointers: A Case For Software Translation on GPUs," ISCA, IEEE, ACM, 2016.

[16] "Threaded I/O Tester." https://sourceforge.net/p/tiobench.

[17] "GPU Support in Apache Spark and GPU/CPU Mixed Resource Scheduling at Production Scale." http://www.spark.tc/gpu-support-in-spark-and-gpu-cpu-mixed-resource-scheduling-at-production-scale/, 2016.

[18] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "GPUfs: integrating file systems with GPUs," in *ASPLOS'13*, ACM, 2013.

[19] J. Yoo, Y. Won, J. Hwang, S. Kang, J. Choil, S. Yoon, and J. Cha, "Vssim: Virtual machine based ssd simulator," in *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pp. 1–14, IEEE, 2013.

[20] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 266–277, IEEE, 2011.

[21] "A fast GPU memory copy library based on NVIDIA GPUDirect RDMA technology." https://github.com/NVIDIA/gdrcopy, 2015.

[22] "Evacuate struct_page from the block layer." https://lwn.net/Articles/636968/, 2015.

[23] "FOSS4G Benchmark." https://wiki.osgeo.org/wiki/FOSS4G_Benchmark.

[24] "True Marble." http://www.unearthedoutdoors.net/global_data/true_marble/.

[25] VMWare, "vRealize Log Insight." http://www.vmware.com/products/vrealize-log-insight.html.

[26] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis, "Regular expression matching on graphics hardware for intrusion detection," in *International Workshop on Recent Advances in Intrusion Detection*, pp. 265–283, Springer, 2009.

[27] Antonio Torralba, Robert Fergus and William T Freeman, "80 Million Tiny Images: A Large Data Set for Nonparametric Object and Scene Recognition," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 30, no. 11, pp. 1958–1970, 2008.

[28] "Benchmarking GPUDirect RDMA on Modern Server Platforms." https://devblogs.nvidia.com/parallelforall/benchmarking-gpudirect-rdma-on-modern-server-platforms/, 2014.

[29] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "GPUfs: Integrating a File System with GPUs," *TOCS*, vol. 32, no. 1, p. 1, 2014.

[30] *OpenCAPI*. http://opencapi.org/.

[31] *Cache Coherent Interconnect for Accelerators (CCIX)*. http://www.ccixconsortium.com/.

Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters

Hao Zhang[1,2], Zeyu Zheng[2], Shizhen Xu[1], Wei Dai[1,2], Qirong Ho[2], Xiaodan Liang[1], Zhiting Hu[1,2], Jinliang Wei[1], Pengtao Xie[1,2], Eric P. Xing[2]

Carnegie Mellon University[1], Petuum Inc.[2]

Abstract

Deep learning models can take weeks to train on a single GPU-equipped machine, necessitating scaling out DL training to a GPU-cluster. However, current distributed DL implementations can scale poorly due to substantial parameter synchronization over the network, because the high throughput of GPUs allows more data batches to be processed per unit time than CPUs, leading to more frequent network synchronization. We present Poseidon, an efficient communication architecture for distributed DL on GPUs. Poseidon exploits the layered model structures in DL programs to overlap communication and computation, reducing bursty network communication. Moreover, Poseidon uses a hybrid communication scheme that optimizes the number of bytes required to synchronize each layer, according to layer properties and the number of machines. We show that Poseidon is applicable to different DL frameworks by plugging Poseidon into Caffe and TensorFlow. We show that Poseidon enables Caffe and TensorFlow to achieve 15.5x speed-up on 16 single-GPU machines, even with limited bandwidth (10GbE) and the challenging VGG19-22K network for image classification. Moreover, Poseidon-enabled TensorFlow achieves 31.5x speed-up with 32 single-GPU machines on Inception-V3, a 50% improvement over the open-source TensorFlow (20x speed-up).

1 Introduction

Deep learning (DL) is a class of machine learning (ML) approaches that has achieved notable success across a wide spectrum of tasks, including speech recognition [10], visual recognition [34, 35] and language understanding [21, 20]. These DL models exhibit a high degree of model complexity, with many parameters in deeply layered structures that usually take days to weeks to train on a GPU-equipped machine. The high computational cost of DL programs on large-scale data necessitates the training on distributed GPU cluster in order to keep the training time acceptable.

DL software such as TensorFlow [1] and Caffe [14] allow practitioners to easily experiment with DL models on a single machine. However, their distributed implementations can scale poorly for larger models. For example, we find that on the VGG19-22K network (229M parameters), open-source TensorFlow on 32 machines can be slower than single machine (Section 5.1). This observation underlines the challenge of scaling DL on GPU clusters: the high computational throughput of GPUs allows more data batches to be processed per minute (than CPUs), leading to more frequent network synchronization that grows with the number of machines. Existing communication strategies, such as parameter servers (PS) for ML [31, 19], can be overwhelmed by the high volume of communication [7]. Moreover, despite the increasing availability of faster network interfaces such as Infiniband or 40GbE Ethernet, GPUs have continued to grow rapidly in computational power, and continued to produce parameter updates faster than can be naively synchronized over the network. For instance, on a 16-machine cluster with 40GbE Ethernet and one Titan X GPU per machine, updates from the VGG19-22K model will bottleneck the network, so that only an 8x speedup over a single machine is achieved (Section 5.1).

These scalability limitations in distributed DL stem from at least two causes: (1) the gradient updates to be communicated are very large matrices, which quickly saturate network bandwidth; (2) the iterative nature of DL algorithms causes the updates to be transmitted in bursts (at the end of an iteration or batch of data), with significant periods of low network usage in between. We propose that a solution to these two problems should exploit the structure of DL algorithms on two levels: on one hand, it should identify ways in which the matrix updates can be separated from each other, and then schedule them in a way that avoids bursty network traffic. On the other hand, the solution should also exploit the structure of the matrix updates themselves, and wherever possible, re-

duce their size and thus the overall load on the network. For such a solution to be relevant to practitioners (who may have strong preferences for particular frameworks), we would prefer not to exploit specific traits of TensorFlow's or Caffe's design, but should strive to be relevant to as many existing frameworks as possible.

With this motivation, we design Poseidon, an efficient communication architecture for data-parallel DL on distributed GPUs. Poseidon exploits the sequential layer-by-layer structure in DL programs, finding independent GPU computation operations and network communication operations in the training algorithm, so that they can be scheduled together to reduce bursty network communication. Moreover, Poseidon implements a hybrid communication scheme that accounts for each DL program layer's mathematical properties as well as the cluster configuration, in order to compute the network cost of different communication methods, and select the cheapest one – currently, Poseidon implements and supports a parameter server scheme [31] that is well-suited to small matrices, and a sufficient factor broadcasting scheme [32] that performs well on large matrices. We focus on synchronous parallel training which is shown to yield faster convergence compared with asynchronous training in distributed DL (as measured by wall clock time) on GPUs [7, 2]. Unless otherwise specified, our discussion in this paper assumes synchronous replication of model parameters in each training iteration, although we note that Poseidon's design can easily be applied to asynchronous or bounded-asynchronous consistency models [12, 8].

To demonstrate Poseidon's applicability to multiple DL frameworks, we implement it into two different DL frameworks: Caffe and TensorFlow, and show that Poseidon allows them to scale almost-linearly in algorithm throughput with additional machines, while incurring little additional overhead even in the single machine setting. For distributed execution, with 40GbE network bandwidth available, Poseidon consistently delivers near-linear increases in throughput across various models and engines: 31.5x speedup on training the Inception-V3 network using TensorFlow engine on 32 nodes, which improves 50% upon the original TensorFlow (20x); when training a 229M parameter network (VGG19-22K), Poseidon still achieves near-linear speedup (30x on 32 nodes) using both Caffe and TensorFlow engines, while distributed TensorFlow sometimes experiences negative [37] scaling with additional machines. Our experiments also confirm that Poseidon successfully alleviates network communication bottlenecks, by reducing the required bandwidth for parallelizing large models. For example, when training VGG19-22K under limited bandwidth (10GbE), in contrast to a PS-based parallelization which only achieves 4x speedup with 16 machines, Poseidon effectively reduces the communication overheads by automatically specializing the best communication method for each layer, and is able to keep linearly scaling with throughput. Compared to other communication reduction methods [4, 36], Poseidon demonstrates either systems advantages (increased algorithm throughput) or statistical advantages (fewer algorithm steps or iterations to reach a fixed termination criteria). Poseidon does not suffer much from imbalanced communication loads, which we found to be the case when using the sufficient factor strategy used in Project Adam [4]. Poseidon also guarantees that the number of algorithm steps to reach termination remains unchanged, unlike the 1-bit quantization strategy used in CNTK [36] which is approximate and can hurt statistical performance in some applications.

The rest of the paper is organized as follows. Section 2 motivates Poseidon with introduction on large-scale DL, parameter servers and sufficient factor broadcasting. Section 3 and section 4 elaborates Poseidon's design and implementation, respectively. Section 5 evaluates Poseidon by training different models over multiple datasets, including comparisons to state-of-the-art GPU-based distributed DL systems. Section 6 discusses related works and section 7 concludes.

2 Large-scale Deep Learning

In this section, we formulate the DL training as an iterative-convergent algorithm, and describe parameter server (PS) and sufficient factor broadcasting (SFB) for parallelizing such computation on clusters.

2.1 Distributed Deep Learning

DL programs are distinguished from other ML programs mainly by their use of neural networks (NNs), a family of hierarchical models containing many layers, from as few as 5-10 [16] to as many as 100s [11]. Figure 1 illustrates a neural network with 6 layers. The first layer (green) is an input layer that reads data in application-specific formats, e.g., raw pixels if it is trained to classify images. The input layer is connected to a sequence of intermediate layers (cyan, orange), each of which consists of a few neurons, where each neuron applies a function transformation f on its input and produces an output. A vector output is obtained by concatenating the output of all neurons from a layer. By stacking multiple intermediate layers, the NN can transform raw input data one layer at a time, first into a series of intermediate representations, and finally into the desired output or prediction (red). DL programmers usually need to specify the computation of a layer by defining two properties of its neurons. The first is the transformation function $f(W,x)$, where x is the input to the neuron, and W is an *optional trainable* parameter. The other is the connectivity that determines how the neuron should be connected to its adjacent layer. For

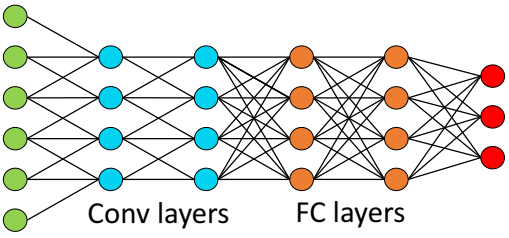


Figure 1: A convolutional neural network with 6 layers.

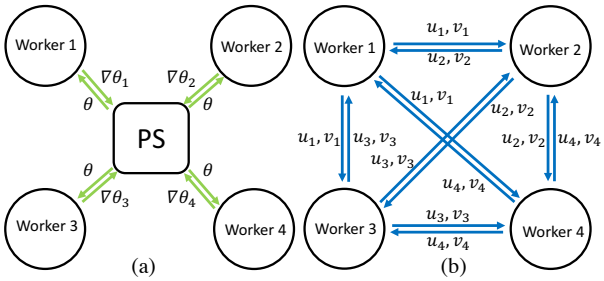


Figure 2: An illustration of (a) the parameter server and (b) sufficient factor broadcasting for distributed ML.

instance, a convolutional neural network has two types of neuron: convolutional (CONV) neuron (cyan) that are only locally connected to a subset of neurons in its previous layer, and fully-connected (FC) neurons (orange).

Most NNs need to be trained with data to give accurate predictions. Stochastic gradient descent (SGD) and backpropagation are commonly employed to train NNs iteratively – each iteration performs a feed forward (FF) pass followed with a backpropagation (BP) pass. In the FF pass, the network takes a training sample as input, forwards from its input layer to output layer to produce a prediction. A loss function is defined to evaluate the prediction error, which is then backpropagated through the network in reverse, during which the network parameters are updated by their gradients towards where the error would decrease. After repeating a sufficient number of passes, the network will usually converge to some state where the loss is close to a minima, and the training is then terminated. In a mathematical form, given data D and a loss function $\mathcal{L}$, fitting the parameters θ of a NN can be formulated as an *iterative-convergent* algorithm that repeatedly executing the update equation

$$\theta^{(t)} = \theta^{(t-1)} + \varepsilon \cdot \nabla_{\mathcal{L}}(\theta^{(t-1)}, D^{(t)}) \quad (1)$$

until θ reaches some stopping criteria, where t denotes the iteration. The update function $\nabla_{\mathcal{L}}$ calculates the gradients of $\mathcal{L}$ over current data $D_i (D_i \in D)$. The gradients are then scaled by a learning rate ε and applied on θ as updates. As the gradients are additive over data samples i, i.e., $\theta^{(t)} = \theta^{(t-1)} + \varepsilon \cdot \sum_i \nabla_{\mathcal{L}}(\theta^{(t-1)}, D_i)$, for efficiency, we usually feed a batch of training samples $D^{(t)} (D^{(t)} \subset D)$ at each training iteration t, as in Eq.1.

In large-scale deep learning, data D are usually too large to process on a single machine in acceptable time. To speedup the training, we usually resort to *data parallelism*, a parallelization strategy that partitions the data D and distributes to a cluster of computational worker machines (indexed by $p = 1, \cdots, P$), as illustrated in Figure 2. At each iteration t, every worker fetches a batch $D_p^{(t)}$ from its data partition and computes the gradients $\nabla_{\mathcal{L}}(\theta^{(t)}, D_p^{(t)})$. Gradients from all workers are then aggregated and applied to update $\theta^{(t)}$ to $\theta^{(t+1)}$ following

$$\theta^{(t+1)} = \theta^{(t)} + \varepsilon \sum_{p=1}^{P} \nabla_{\mathcal{L}}(\theta^{(t)}, D_p^{(t)}) \quad (2)$$

Data-parallelism allows data to be locally partitioned to each worker, which is advantageous for large datasets. It however requires every worker to have read and write access to the shared model parameters θ, which causes communication among workers; this shared access can be provided by a parameter server architecture [31, 4] (Figure 2a) or a peer-to-peer broadcasting architecture [32] (Figure 2b), both are designed for general-purpose data-parallel ML programs on CPUs.

Parameter Server. A parameter server (PS) is a distributed shared memory system that provides systematic abstraction of iterative-convergent algorithms in data-parallel distributed ML. Typically, PS enables each worker to access the global model parameters θ via network communications following the client-server scheme. DL can be trivially parallelized over distributed workers using PS with the following 3 steps: (1) Each worker computes the gradients ($\nabla_{\mathcal{L}}$) on their own data partition and send them to remote servers; (2) servers receive the updates and apply ($+$) them on globally shared parameters; (3) a consistency scheme coordinates the synchronization among servers and workers (Figure 2a).

Sufficient Factor Broadcasting. Many ML models represent their parameters θ as matrices. For example, fully-connected NNs, when trained using SGD, their gradient $\nabla\theta$ over a training sample is a rank-1 matrix, which can be cast as the outer product of two vectors u, v: $\nabla\theta = uv^\top$, where u and v are called *sufficient factors* (SFs). Sufficient factor broadcasting (SFB) [32] is designed to parallelize these models by broadcasting SFs among workers and then reconstructing the gradient matrices $\nabla\theta$ using u, v locally. SFB presents three key differences from PS: (1) SFB uses a P2P communication strategy that transmits SFs instead of full matrices. (2) Unlike gradients, SFs are not additive over training samples, i.e., the number of SFs needed to be transmitted grows linearly with the number of data samples (not data batches); (3) the overall communication overheads of SFB increase quadratically with the number of workers.

2.2 Parallel DL on Distributed GPUs

Modern DL models are mostly trained using NVIDIA GPUs, because the primary computational steps (e.g., matrix-matrix multiplications) in DL match the SIMD operation that could be efficiently performed by GPUs.

In practice, DL practitioners often use single-node software frameworks, such as Caffe [14] and Torch [6], which mathematically derive the correct training algorithm and execute it on GPU by calling GPU-based acceleration libraries, such as CUBLAS and cuDNN. It is thus straightforward to parallelize these programs across distributed GPUs using either PS or SFB, by moving the computation from CPU to GPU, and performing memory copy operations (between DRAM and GPUs) or communication (among multiple nodes) whenever needed. However, we argue below and show empirically in Section 5 that these usually lead to suboptimal performance.

The inefficiency is mainly caused by parameter synchronization via the network. Compared to CPUs, GPUs are an order of magnitude more efficient in matrix computations; the production of gradients on GPUs is much faster than they can be naively synchronized over the network. As a result, the training computations are usually bottlenecked by communications. For example, when training AlexNet [16] (61.5M parameters) on Titan X with a standard batch size 256, 240 million gradients will be generated per second on each GPU (0.25s/batch). If we parallelize the training on 8 nodes using a PS, with every node also holding $1/8$ of parameters as a PS shard; then, every node needs to transfer $240\text{M}\times 7/8\times 4 = 840\text{M}$ float parameters in one second to make sure the next iteration of computation not being blocked. Apparently, the demanded throughput (>26Gbps) exceeds the bandwidth that commodity Ethernet (i.e., 1GbE and 10GbE Ethernet) provides; the GPUs distributed across clusters cannot be fully utilized. Practically, it is usually difficult to partition the parameters completely equally, which will result in more severe bandwidth demands, or bursty communication traffic on several server nodes (as we will show in Section 5.3), which prevents the trivial realization of efficient DL on distributed GPUs [1]. We next describe our strategies and system design to overcome the aforementioned obstacles.

3 Poseidon Design

In this section, we first analyze the DL program in both a single-node and distributed environment by decomposing the program into a sequence of operations. Based on it, we introduce two strategies to address the issues.

The Structure of DL Programs. At the core of the DL program is the BP algorithm that performs forward-backward pass through the network repeatedly. If we define a forward and a backward pass through the lth layer of a network as f_t^l and b_t^l, respectively, then a **C**omputation step at iteration t is notated as $C_t = [f_t^1, \cdots, f_t^L, b_t^L, \cdots, b_t^1]$, as illustrated in Fig. 3(a). When executing on distributed GPUs, inter-machine communications are required after each C step to guarantee the synchronized replication of model parameters. We similarly define the **S**ynchronization step S_t as the process that a worker sends out locally generated updates and then receives updated parameters from remote workers at iteration t. Therefore, a naive parallelization of DL training over distributed GPUs using either PS or SFB can be expressed as alternating C_t and S_t defined above. We note that DL training is highly sequential; the communication and computation perform sequentially, waiting each other to finish (Fig. 3a).

[1]Frequent memory copy operations between DRAM and GPU memory can also cause extra overheads, which is minor compared to the network communication according to our empirical results. However, our strategies in this paper can also alleviate this overhead.

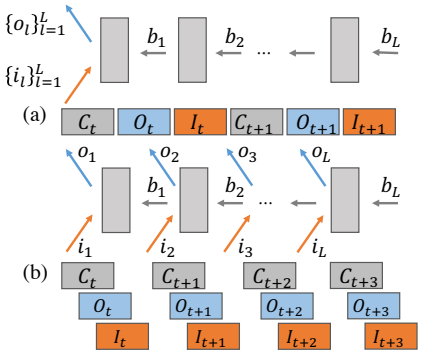


Figure 3: (a) Traditional backpropagation and (b) wait-free backpropagation on distributed environment.

Fortunately, we also note that as every layer of a NN contains an independent set of parameters, S_t can be decoupled as $S_t = (s_t^1, \cdots, s_t^L)$, by defining s_t^l as the synchronization of parameters of layer l. If we further decompose $s_t^l = [o_t^l, i_t^l]$ as first sending out local updates of layer l (o_t^l) and reads in the updated parameters remotely (i_t^l), we can rewrite a training iteration as: $[C_t, S_t] = [f_t^1, \cdots, f_t^L, b_t^L, \cdots, b_t^1, o_t^L, \cdots, o_t^1, i_t^L, \cdots, i_t^1]$. The sequential nature of the BP algorithm presents us an opportunity to overlap the computations and communications. Our first strategy, *wait-free backpropagation*, overlaps C_t and S_t by partially rescheduling those b_t and s_t that are independent. Our second strategy, *hybrid communication*, utilizes the independency among s_t, and tries to reduce the communication overheads by specializing different communication methods for different s_t.

3.1 Wait-free Backpropagation

The wait-free backpropagation (WFBP) is designed to overlap communication overheads with the computation based on two key independencies in the program: (1) the send-out operation o_t^l is independent of backward operations $b_t^i (i < l)$, so they could be executed concurrently without blocking each other; (2) the read-in operation i_t^l could update the layer parameters as long as b_t^l was finished, without blocking the subsequent backward operations $b_t^i (i < l)$. Therefore, we can enforce each layer l to start its communication once its gradients are generated after b_t^l, so that the time spent on operation s_t^l could be overlapped with those of $b_t^i (i < l)$, as shown in Fig. 3b.

WFBP is most beneficial for training DL models that have their parameters concentrating at upper layers (FC layers) but computation concentrating at lower layers (CONV layers)[2], e.g., VGG [26] and AdamNet [4, 7]), because it overlaps the communication of top layers (90% of communication time) with the computation of bottom layers (90% of computation time) [37, 7]. Besides chain-like NNs, WFBP is generally applicable to other non-chain like structures (e.g., tree-like structures), as the parameter optimization for deep neural networks depends on adjacent layers (and not the whole network), there is always an opportunity for parameter optimization (i.e., computation) and communication from different layers to be performed concurrently.

Some DL frameworks, such as TensorFlow, represent the data dependencies of DL programs using graphs, therefore implicitly enable auto-parallelization. However, they fail on exploring the potential opportunities of parallelization between iterations. For example, TensorFlow needs to fetch the updated parameters from the remote storage at the beginning of each iteration, while it is possible to overlap this communication procedure with the computation procedure of the previous iteration. In comparison, WFBP enforces this overlapping by explicitly pipelining compute, send and receive procedures. We describe our implementation of WFBP in Section 4 and empirically show its effectiveness in Section 5.1.

3.2 Hybrid Communication

While WFBP overlaps communication and computation, it does not reduce the communication overhead. In situations where the network bandwidth is limited (e.g., commodity Ethernet or the Ethernet is shared with other communication-heavy applications), the communication would still be unacceptably slow. To address the issue, we introduce a *hybrid communication* (HybComm) strategy that combines the best of PS and SFB by being aware of both the mathematical property of DL models and the structure of computing clusters. Our idea comes from two observations: first, as presented in Section 3, the synchronization operations $\{S_t^l\}_{l=1}^L$ are independent of each other, meaning that we can use different communication methods for different S_t^l by specializing o_t^l and i_t^l according to the two methods described in Figure 2; second, a NN structure is usually predefined and fixed throughout the training – by measuring the number of parameters needed to transferred, we are able to estimate the communication overhead, so that we can always choose the optimal method even before the communication happens.

Consider training VGG19 network [26], the overheads of S_t^l could be estimated as follows (Table 1): assume the batch size $K = 32$, the number of workers and server nodes $P_1 = P_2 = 8$ (assume parameters are equally partitioned over all server shards), respectively. On one hand, if l is an FC layer (with shape $4096 \times 4096, M = N = 4096$), synchronizing its parameters via PS will transfer $2MN \approx 34$ million parameters for a worker node, $2P_1MN/P_2 \approx 34$ million for a server node, and $2MN(P_1+P_2-2)/P_2 \approx 58.7$ million for a node that is both a server and a worker, compared to $2K(M+N)(P_1-1) \approx 3.7$ million for a single node using SFB. On the other hand, if l is a CONV layer, the updates are indecomposable and sparse, so we can directly resort to PS. Therefore, the synchronization overheads depend not only on the model (type, shape, size of the layer), but also the size of the clusters. The optimal solution usually changes with M, N, K, P_1, P_2. HybComm takes into account these factors and allows to dynamically adjust the communication method for different parts of a model – it always chooses the best method from available ones whenever it results in fewer communication overheads.

Method	Server	Worker	Server & Worker
PS	$2P_1MN/P_2$	$2MN$	$2MN(P_1+P_2-2)/P_2$
SFB	**N/A**	$2K(P_1-1)(M+N)$	**N/A**
Adam (max)	$P_1MN+P_1K(M+N)$	$K(M+N)+MN$	$(P_1-1)(MN+KM+KN)$

Table 1: Estimated communication cost of PS, SFB and Adam for synchrnizing the parameters of a $M \times N$ FC layer on a cluster with P_1 workers and P_2 servers, when batchsize is K.

Microsoft Adam [4] employs a different communication strategy from those in Figure 2. Instead of broadcasting SFs across workers, they first send SFs to a parameter server shard, then pull back the whole updated parameter matrices. This seems to reduce the total number of parameters needed to be communicated, but usually leads to load imbalance; the server node that holds the corresponding parameter shard overloads because it has to broadcast the parameter matrices to all workers ($P_1MN + P_1K(M+N)$ messages need to be broadcasted), which easily causes communication bottleneck (Section 5.3). It is noticeable that reconstructing gradients from SFs may cause extra computation cost, which however is often negligible compared to communication. We describe our implementation of HybComm in the next section, and assess its effectiveness in Section 5.

4 Implementation

This section first elaborates Poseidon's system architecture and APIs, and then describes how to modify a framework using Poseidon to enable distributed execution.

4.1 System Implementation and APIs

Figure 4 illustrates the architecture of Poseidon: a C++ communication library that manages parameter communication for DL programs running on distributed GPUs. It has three main components: coordinator, that main-

[2]Most classification models will fall into this family if the number of classes to be classified is large.

Method	Owner	Arguments	Description
`BestScheme`	Coordinator	A layer name or index	Get the best communication scheme of a layer
`Query`	Coordinator	A list of property names	Query information from coordinators' information book
`Send`	Syncer	None	Send out the parameter updates of the corresponding layer
`Receive`	Syncer	None	Receive parameter updates from either parameter server or peer workers
`Move`	Syncer	A GPU stream and an indicator of move direction	Move contents between GPU and CPU, do transformations and application of updates if needed
`Send`	KV store	updated parameters	Send out the updated parameters
`Receive`	KV store	parameter buffer of KV stores	Receive gradient updates from workers

Table 2: Poseidon APIs for parameter synchronization.

Algorithm 1 Get the best comm method of layer l

1: **function** BESTSCHEME(l)
2: $layer_property$ = Query(l.name)
3: P_1, P_2, K = Query('n_worker', 'n_server', 'batchsize')
4: **if** $layer_property$.type == 'FC' **then**
5: $M = layer_property$.width
6: $N = layer_property$.height
7: **if** $2K(P_1-1)(M+N) \leq \frac{2MN(P_1+P_2-2)}{P_2}$ **then**
8: return 'SFB'
9: **end if**
10: **end if**
11: return 'PS'
12: **end function**

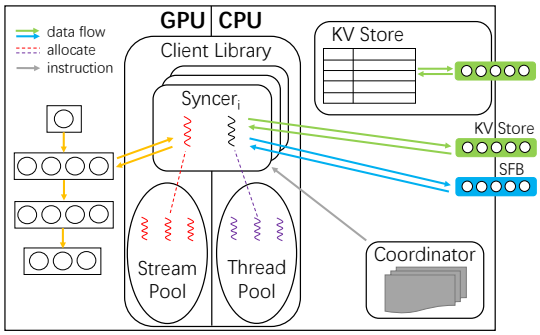


Figure 4: An overview of the architecture of Poseidon.

tains the model and the cluster configuration; KV store, a shared memory key-value store that provides support for parameter server based communication; client library, which is plugged into DL programs to handle parameter communication. Their APIs are listed in Table 2.

Coordinator. To setup distributed training, the client program (e.g., Caffe) first instantiates Poseidon by creating a coordinator within its process. Coordinators will first collect necessary information, including the cluster information (e.g., the number of workers and server nodes, their IP addresses) and the model architecture (e.g., the number of layers, layer types, number of neurons and how they are connected, etc.). With the information, the coordinator will initialize the KV stores and the client library with two steps: (1) allocate proper communication ports for each PS shard and peer worker; (2) determine what parameters should be transmitted via the KV store and what by SFB, and hash the parameters equally to each KV store if necessary, and save the mapping in the information book, which, throughout the whole training, is maintained and synchronized across nodes, and could be accessed elsewhere through coordinator's `Query` API. Besides, the coordinator provides another API `BestScheme` that takes in a layer and returns the optimal communication scheme for it according to the strategy described in Section 3.2 (Algorithm 1).

KV Store. The KV store is implemented based on a bulk synchronous parameter server [31, 7], and instantiated by coordinators on a list of user-specified "server" machines. Each instance of the KV store holds one shard of the globally shared model parameters in the form of a set of KV pairs, of which each KV pair is stored on a chunk of DRAM. Poseidon sets the size of a KV pair to a fixed small size (e.g., 2MB), so as to partition and distribute model parameters to server nodes as equally as possible, reducing the risk of Ethernet bottleneck. Each KV store instance manages a parameter buffer on RAM, and provides PS-like APIs, such as `Receive` and `Send`, for receiving and applying updates from client libraries, or sending out parameters. It will regularly checkpoint current parameter states for fault tolerance.

Client Library. Poseidon coordinates with DL programs via its client library. Particularly, users plug the client library into their training program, and the client library will create a *syncer* for each NN layer during network assembling (so that each layer one-to-one maps to one syncer), accounting for its parameter synchronization. Each sycner is then initialized, for example, setting up connections to its corresponding PS shards or (remote) peer syncers according to the coordinator's information book, and allocating a small memory buffer for receiving remote parameter matrices or SFs, etc.

The client library manages a CPU thread pool and a GPU stream pool on the worker machine, which can be allocated by the syncer APIs when there is a syncer job created. The syncer has three main APIs, `Send`, `Receive` and `Move`, to be used in client programs. The `Move` API takes care of the memory movement between RAM and GPU memory, and performs necessary computation, e.g., the transformation between SFs and gradients, and the application of updates. It is multi-threaded using the CUDA asynchronous APIs, and will trigger an allocation from the client library's thread/stream pools when a syncer job starts (see L14 of Algorithm 2). The `Send` and `Receive` are communication APIs that synchronize layer parameters across different model repli-

cas. The `Send` API is nonblocking; it sends out parameter updates during backpropagation once they are generated, following the protocol returned by coordinator's `BestScheme` API. The `Receive` API will be called once `Send` is finished. It requests either fresh parameter matrices from the KV stores or SFs from its peer syncers, and will block its current thread until it receives all of what it requested. The received messages are put into the syncer's memory buffer for the `Move` API to fetch.

Managing Consistency. Poseidon implements the bulk synchronous consistency (BSP) model as follows. The client library maintains a binary vector C with length the number of syncers and values reset to zeros at the start of each iteration. A syncer will set its corresponding entry in C as 1 when its job finishes, and the client starts next iteration when all entries are 1. While, the KV store maintains a zero-initialized count value for each KV pair at the start of each iteration. Every time when there is an update being applied on a KV pair, its count value is increased by 1. The KV pair will be broadcasted via its `Send` API when its count equals to the number of workers. Poseidon handles stragglers by simply dropping them. Although asynchronous models can alleviate the straggler problem in distributed ML [12], Poseidon focuses on synchronous parallel training, because synchronous execution yields the fastest per-iteration improvement in accuracy for distributed DL (as measured by wall clock time) on GPUs [7, 2] (see Section 5.1).

Algorithm 2 Parallelize a DL library using Poseidon

```
1:  function TRAIN(net)
2:      for iter = 1 → T do
3:          sync_count = 0
4:          net.Forward()
5:          for l = L → 1 do
6:              net.BackwardThrough(l)
7:              thread_pool.Schedule(sync(l))
8:          end for
9:          wait_until(sync_count == net.num_layers)
10:     end for
11: end function
12: function SYNC(l)
13:     stream = stream_pool.Allocate()
14:     syncers[l].Move(stream, GPU2CPU)
15:     syncers[l].method = coordinator.BestScheme(l)
16:     syncers[l].Send()
17:     syncers[l].Receive()
18:     syncers[l].Move(stream, CPU2GPU)
19:     sync_count++
20: end function
```

4.2 Integrate Poseidon with DL Libraries

Poseidon could be plugged into most existing DL frameworks to enable efficient distributed execution. Algorithm 2 provides an example. Specifically, one needs to first include Poseidon's client library into the framework, then figure out where the backpropagation proceeds (L6), and insert Poseidon's syner APIs in between gradient generation and application (L7). We demonstrate in Section 5.1 that with slight modifications (150 and 250 LoC for Caffe and TensorFlow), both Poseidon-enable Caffe and TensorFlow deliver linear scalings up to 32 GPU machines. Poseidon respects the programming interfaces by the native DL library and stores necessary arguments for distributed execution as environment variables to allow zero changes on the DL application programs.

5 Evaluation

In this section, we evaluate Poseidon's performance on scaling up DL with distributed GPUs. We focus on the image classification task where DL is most successfully applied. Our evaluation reveals the following results: (1) Poseidon has little overhead when plugged into existing frameworks; it achieves near-linear speedups across different NNs and frameworks, on up to 32 Titan X-equipped machines. (2) Poseidon's system design effectively improves GPU and bandwidth utilization. (3) Poseidon's communication strategy HybComm effectively alleviates the communication bottleneck, thus achieves better speedups under limited bandwidth; Moreover, Poseidon compares favorably to other communication-reduction methods, such as the SF strategy in Adam [4], and the 1-bit quantization in CNTK [36].

Cluster Configuration. We conduct our experiments on a GPU cluster with each node equipped with a NVIDIA GeForce TITAN X GPU card, an Intel 16-core CPU and 64GB RAM, interconnected via a 40-Gigabit Ethernet switch. All cluster nodes have shared access to a NFS and read data through the Ethernet interface. We run our system on UBUNTU 16.04, with NVIDIA driver version 361.62, CUDA 8.0 and cuDNN v5.

Computation Engines. We deploy Poseidon on two DL frameworks, Caffe [14] and TensorFlow [1]. For Caffe, we use the official version at 2016/06/30 as the single node baseline, and modify it using Poseidon's client library API for distributed execution. For TensorFlow, we use its open source version r0.10, and parallelize its single-node version with Poseidon's client library, and compare to its original distributed version. [3]

Dataset and Models. Our experiments use three well-known image classification datasets. (1) CIFAR-10 [15], which contains 32×32 colored images of 10 classes, with 50K images for training and 10K for testing; (2) ILSVRC12 [23], a subset of ImageNet22K that has 1.28 million of training images and 50K validation images in 1,000 categories; (3) ImageNet22K [23], the largest public dataset for image classification, including 14,197,087

[3] Note that as the distributed engine of TensorFlow is highly optimized (e.g., auto-parallelization of graphs [1]). Poseidon avoids leveraging any build-in optimization of distributed TensorFlow by parallelizing its single-node version instead.

Model	# Params	Dataset	Batchsize
CIFAR-10 quick	145.6K	CIFAR10	100
GoogLeNet	5M	ILSVRC12	128
Inception-V3	27M	ILSVRC12	32
VGG19	143M	ILSVRC12	32
VGG19-22K	229M	ImageNet22K	32
ResNet-152	60.2M	ILSVRC12	32

Table 3: Neural networks for evaluation. Single-node batchsize is reported. The batchsize is chosen based on the standards reported in literature (usually the maximum batch size that can fill in the GPU memory).

labeled images from 21,841 categories.

We test Poseidon's scalability across different neural networks: (1) CIFAR-10 quick: a toy CNN from Caffe that converges at 73% accuracy for classifying images in CIFAR-10 dataset; (2) GoogLeNet [27]: a 22-layer CNN with 5M parameters. (3) Inception-V3 [28]: the ImageNet winner, an improved version of GoogLeNet from TensorFlow; (4) VGG19: A popular feature extraction network in the computer vision community [26] that has 16 CONV layers and 3 FC layers, in total 143M parameters; (5) VGG19-22K: we modify the VGG19 network by replacing its 1000-way classifier with a 21841-way classifier, to classify images from the ImageNet22K dataset. The modified network has 229M parameters. (6) ResNet-152: the ImageNet winner network with 152 layers. We list their statistics and configurations in Table 3.

Metrics. In this paper, we mainly focus on metrics that measure the system performance, such as speedups on throughput (number of images scanned per second). Our experiments focus on medium-scale distributed cluster with up to 32 machines, which distributed DL empirically benefits most from. Larger clusters require larger batch sizes, which hurt the convergence rate of each iteration [3, 7]. For completeness, we also report the statistical performance (time/epoch to converge) on ResNet-152. Poseidon uses synchronized replication which enables many models to converge in fewer steps [1, 7, 3, 2].

5.1 Scalability

To demonstrate Poseidon's scalability, we train CNNs using Poseidon with different computational engines, and compare different systems in terms of their speedups on throughput. For Caffe engine, we train GoogLeNet VGG19 and VGG19-22K networks; for TensorFlow engine, we train Inception-V3, VGG-19, VGG19-22K.

Caffe Engine. Figure 5 shows the throughput vs. number of workers when training the three networks using Caffe engine, given 40GbE Ethernet bandwidth available. We compare the following systems: (1) *Caffe*: unmodified Caffe that executes on a single GPU; (2) *Caffe+PS*: we parallelize Caffe using a vanilla PS, i.e., the parameter synchronization happens sequentially after the backpropagation in each iteration; (3) *Caffe+WFBP*: Parallelized Caffe using Poseidon so the communication and computation are overlapped. However, we disable HybComm so that parameters are synchronized only via PS; (4) *Poseidon*: the full version of Poseidon-Caffe.

Poseidon shows little overheads when combined with Caffe; running on a single node with no communication involved, Poseidon-Caffe can process 257, 35.5 and 34.2 images per second when training GoogLeNet, VGG19 and VGG19-22K, respectively, as compared to the original Caffe, which can process 257, 35.5 and 34.6 images, and Caffe+PS, which can only process 213.3, 21.3 and 18.5 images per second, due to the overheads caused by memory copy operations between RAM and GPU, which have been overlapped by Poseidon with the computation. In distributed environment, the rescheduling of computation and communication significantly improves the throughput: when training GoogLeNet and VGG19, incorporating WFBP achieves almost linear scalings up to 32 machines, and for the larger VGG19-22K network, Caffe+WFBP achieves 21.5x speedup on 32 machines. We conclude that rescheduling and multi-threading the communication and computation are key to the performance of distributed DL on GPUs, even when the bandwidth resource is abundant. Poseidon provides an effective implementation to overlap these operations for DL frameworks, to guarantee better GPU utilization.

When the available bandwidth is sufficient, Poseidon's HybComm strategy shows small improvement on training GoogLeNet and VGG19. However, when training VGG19-22K which has three FC layers that occupy 91% of model parameters, it improves over Caffe-WFBP from 21.5x to 29.5x on 32 nodes.

TensorFlow Engine. We also modify TensorFlow using Poseidon, and compare the following systems in terms of speedup on throughput: (1) *TF*: TensorFlow with its original distributed executions; (2) *TF+WFBP*: we modify TensorFlow using Poseidon's client library. Specifically, we change the *assign* operator in TensorFlow, so that instead of being applied, the parameter updates will be synchronized via *Poseidon's PS interface with WFBP*; (3) *Poseidon*: the full version of Poseidon-parallelized TensorFlow with HybComm enabled.

We train Inception-V3, VGG19 and VGG19-22K models and report the results in Figure 6. Running on a single node, Poseidon processes 43.2, 38.2 and 34.5 images per second on training Inception-V3, VGG19 and VGG19-22K, while original TensorFlow processes 43.2, 38.5 and 34.8 images per second on these three models, respectively – little overhead is introduced by our modification. In distributed execution, Poseidon achieves almost linear speedup on up to 32 machines. Distributed TensorFlow, however, demonstrates only 10x speedup on training Inception-V3 and even fails to scale on training the other two networks in our experiments. To investigate the problem of TensorFlow and explain how Posei-

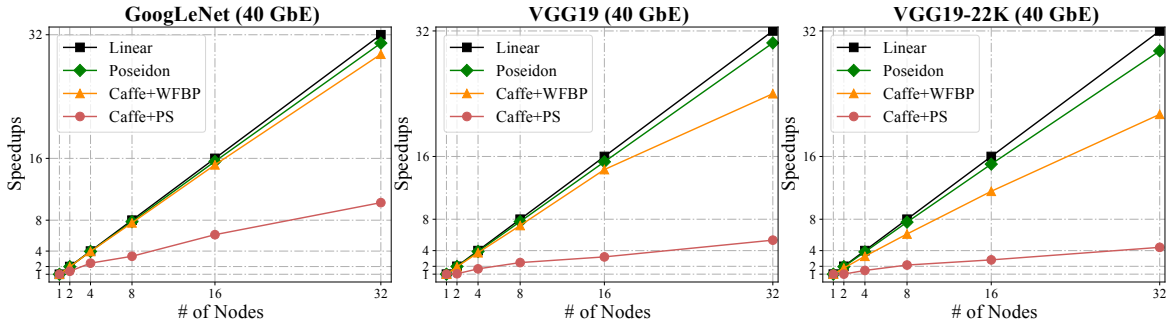


Figure 5: Throughput scaling when training GoogLeNet, VGG19 and VGG19-22K using Poseidon-parallelized Caffe and 40GbE bandwidth. Single-node Caffe is set as baseline (i.e., speedup = 1).

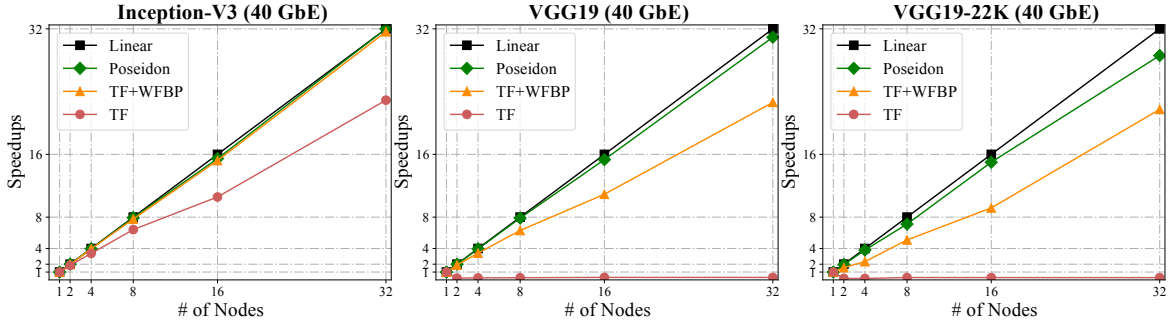


Figure 6: Throughput scaling when training Inception-V3, VGG19 and VGG19-22K using Poseidon-parallelized TensorFlow and 40GbE bandwidth. Single-node TensorFlow is set as baseline (i.e., speedup = 1).

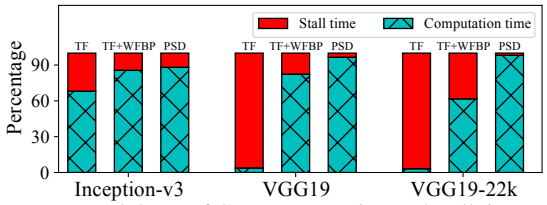


Figure 7: Breakdown of GPU computation and stall time when training the three networks on 8 nodes using different systems.

don improves upon it, we illustrates in Figure 7 the (averaged) ratio of busy and stall time of a GPU when training the three networks using different systems on 8 nodes. Observe that Poseidon keeps GPUs busy in most of the time, while TensorFlow wastes much time on waiting for parameter synchronization. The inefficiency of distributed TensorFlow stems from two sources. First, TensorFlow partitions model parameters in a coarse-grained granularity – each tensor (instead of a KV pair) in the model is assigned to a PS shard. A big tensor (such as the parameter matrix in VGG19) is highly likely to create communication bottleneck on its located server node. Poseidon fixes this problem by partitioning parameters among server nodes in a finer-grained granularity using KV pairs, so that every node has evenly distributed communication load; as an evidence, TF-WFBP demonstrates higher computation-to-stall ratio in Figure 7. Second, TensorFlow cannot reduce the communication overheads while Poseidon's HybComm effectively reduces the size of messages. As a result, Poseidon further improves upon TF-WFBP from 22x to 30x on 32 nodes.

Multi-GPU Settings. Poseidon's key strategies can be directly extended to support distributed multi-GPU environment with minor modifications. Specifically, when there are more than 1 GPU on a worker node, Poseidon will first collect the gradient updates following WFBP locally (either by full matrices or SFs) from multiple GPUs to a leader GPU using *CudaMemcpy(DeviceToDevice)* API. If those updates are determined to be communicated via full matrices, Poseidon will aggregate them locally before sending out. Using Caffe engine on a single node, Poseidon achieves linear scalings on up to 4 Titan X GPUs when training all three networks, outperforming Caffe's multi-GPU version, which shows only 3x and 2x speedups when training GooLeNet and VGG19. When running on AWS p2.8xlarge instances (8 GPUs each node), Poseidon reports 32x and 28x speedups when training GoogLeNet and VGG19 with 4 nodes (32 GPUs in total), confirming our statement that the overheads caused by memory movement between GPUs are usually negligible compared to network communication[4].

Statistical Performance. For completeness, we report in Figure 9 the statistical performance for training ResNet-152 using Poseidon. Poseidon achieves near-linear speedups on both system throughput and statistical convergence: Poseidon delivers 31x speedup in terms of throughput, and reaches 0.24 reported error with less than 90 epochs with both 16 and 32 nodes – thus linear scales in terms of time to accuracy, compared to 8 nodes with batchsize = 32×8, which is a standard set-

[4]The K80 GPUs on p2.8xlarge has less GFLOPS than Titan X used in our main experiments – the communication burden is less severe.

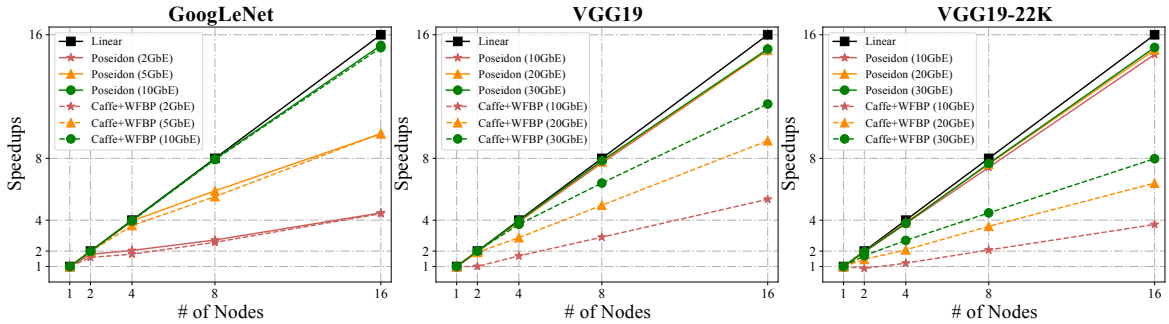


Figure 8: Throughput scaling when training GoogLeNet, VGG19 and VGG19-22K using Poseidon-parallelized Caffe with *varying network bandwidth*. Single-node Caffe is set as baseline (speedup = 1).

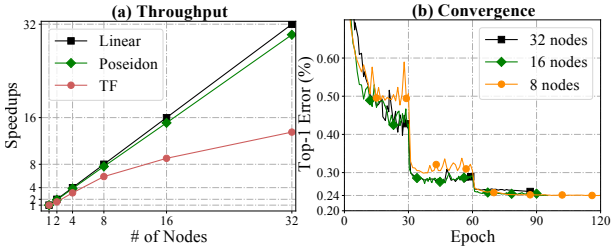


Figure 9: (a) Speedup vs. number of nodes and (b) Top-1 test error vs. epochs for training ResNet-152 using Poseidon-TensorFlow and the original TensorFlow.

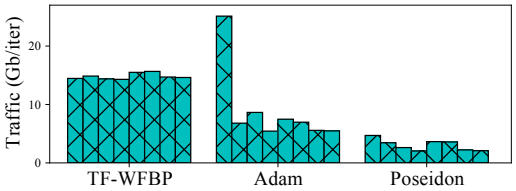


Figure 10: Averaged communication load when training VGG19 using *TF-WFBP*, *Adam* and *Poseidon* with TensorFlow engine. Each bar represents the network traffic on a node.

ting as in [11], echoing recent results that synchronous training on distributed GPUs yields better performance than asynchronous training in terms of time to quality for most NNs [7, 2]. For other NNs in Table. 3, Poseidon delivers the same quality of accuracies as reported in their papers [16, 28, 27, 26] on up to 32 GPUs.

5.2 Bandwidth Experiments

To further assess Poseidon's HybComm strategy, we simulate the environment where network bandwidth is limited. We use Linux traffic control tool *tc* to lower the available bandwidth on each node, and compare the training throughput between with and without HybComm. We focus on Caffe engine in this section because it is lighter and less optimized than TensorFlow.

Figure 8 plots the speedup on throughput vs. number of workers when training GoogLeNet, VGG19 and VGG19-22K with different maximum bandwidth. Clearly, limited bandwidth prevents a standard PS-based system from linearly scaling with number of nodes; for example, given 10GbE bandwidth (which is a commonly-deployed Ethernet configuration in most cloud computing platforms), training VGG19 using PS on 16 nodes can only be accelerated by 8x. This observation confirms our argument that limited bandwidth would result in communication bottleneck when training big models on distributed GPUs. Fortunately, Poseidon significantly alleviates this issue. Under limited bandwidth, it constantly improves the throughput by directly reducing the size of messages needed to be communicated, especially when the batch size is small; when training VGG19 and VGG19-22K, Poseidon achieves near-linear speedup on 16 machines using only 10GbE bandwidth, while an optimized PS would otherwise need 30GbE or even higher to achieve. Note that Poseidon will never underperform a traditional PS scheme because it will reduce to a parameter server whenever it results in less communication overheads; for instance, we observe that Poseidon reduces to PS when training GoogLeNet on 16 nodes, because GoogleNet only has one thin FC layer (1000×1024) and is trained with a large batch size (128).

5.3 Comparisons to Other Methods

In this section, we compare Poseidon against other communication methods, including Adam [4] and CNTK 1-bit quantization [36], and show Poseidon's advantages.

Adam. To save bandwidth, Adam [4] synchronizes the parameters of a FC layer by first pushing SFs generated on all workers to a PS node, and then pulling back the full parameter matrices thereafter. As direct comparisons to Adam [4] are inaccessible, we implement its strategy in Poseidon, and compare it (denoted as *Adam*) to *TF-WFBP* and *Poseidon* by monitoring the network traffic of each machine when training VGG19 on 8 nodes using TensorFlow engine. As shown in Figure 10, the communication workload is highly imbalanced using Adam's strategy. Unlike a traditional PS (TF-WFBP) where the parameters are equally distributed over multiple shards, Adam cannot partition the parameters of FC layers because of their usage of SFs. Although the "push" operation uses SFs to reduce message size, the "pull" requires some server nodes to broadcast big matrices to each worker node, which creates bursty traffic that results in communication bottleneck on them. By contrast, Poseidon either partitions parameters equally over multiple PS shards, or transmits SFs among peer workers, both

are communication load-balanced that avoid bursty communication situations. Quantitatively, Adam delivers 5x speedup with 8 nodes when training VGG19.

CNTK. We compare Poseidon to the 1-bit quantization technique proposed in CNTK [36]. We create a baseline *Poseidon-1bit* which uses the 1-bit strategy to quantize the gradients in FC layers, and add the residual to updates of the next iteration. We then train the CIFAR-10 quick network, and plot the training loss and test error vs. iterations for two systems (both have linear scaling on throughput). As in Figure 11, 1-bit quantization yields worse convergence in terms of accuracy – on 4 GPUs, it achieves 0.5 error after 3K iterations, while Poseidon quickly converges to 0.3 error at iteration 1000. We conjecture this is caused by the quantization residual, which is equivalent to delayed updates that may hurt the convergence performance when training NNs on images, confirmed by [7]. We also directly train VGG19 using CNTK-1bit system, and report 5.8x, 11x, 20x speedups on 8, 16 and 32 nodes, respectively, thus less scale-ups than Poseidon, and also compromised statistical performance due to approximated updates.

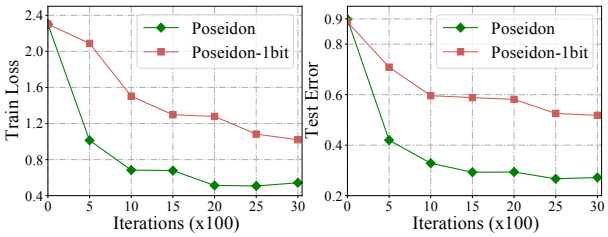


Figure 11: Training loss and test error vs. iteration when training CIFAR-10 quick network using Poseidon and Poseidon-1bit on 4GPUs with Caffe engine.

6 Related Work

PS-based Distributed DL Systems. Based on the parameter server [31, 19] architecture, a number of CPU-based distributed DL systems have been developed, such as [38, 29, 9, 17] and Adam [4]. They are purely PS-based systems on CPU-only clusters, whereas we address the more challenging case of GPU clusters.

Scaling up DL on distributed GPUs is an active field of research. Coates *et al*. [5] build a GPU-based multi-machine system for DL using model parallelism rather than data parallelism, and their implementation is rather specialized for a fixed model structure while demanding specialized hardware, such as InfiBand networking. TensorFlow [1] is Google's distributed ML platform that uses a dataflow graph to represent DL models, and synchronizes model parameters via PS. It therefore cannot dynamically adjust its communication method depending on the layer and cluster information as Poseidon does. MXNet [3] is another DL system that uses PS for distributed execution, and supports TensorFlow-like graph representations for DL models. By auto-parallelizing independent subgraphs, both frameworks implicitly overlap the communication and computation. By contrast, Poseidon has a more explicit way to overlap them via its client library. Hence, Poseidon can be also used to parallelize non-graph-based frameworks. Moreover, both MXNet and TensorFlow do not address the bottleneck caused by limited network bandwidth, which undermines their scalability when training large models with dense layers (e.g., big softmax). Besides, Cui *et al*. propose GeePS [7] that manages the limited GPU memory and report speedups on distributed GPUs. While, GeePS does not address the issue of limited network bandwidth. Therefore, Poseidon's technique could be combined with them to enable better training speedups. Also of note are several efforts to port Caffe onto other distributed platforms, such as SparkNet [22], YahooCaffe [33] and FireCaffe [13], the former reports a 4-5 times speedup with 10 machines (and hence less scalability than our results herein).

Other distributed ML systems. CNTK [36] is a DL framework that supports distributed executions and addresses the problem of communication bottleneck via the 1-bit quantization technique. CNTK demonstrates little negative impact on convergence in speech domains [25, 24]. However, in some other domains (Section 5.3), the performance is usually compromised by noisy gradients [1, 7]. By contrast, Poseidon's HybComm reduces the communication while always guaranteeing synchronous training. There are also growing interest in parallelizing ML applications using peer-to-peer communication, such as MALT [18], SFB [32] and Ako [30]. Poseidon draws inspiration from these works but goes one step further as it is an adaptive best-of-both-worlds protocol, which will select client-server communication whenever it would result in fewer overheads.

7 Conclusion

We present Poseidon, a scalable and efficient communication architecture for large-scale DL on distributed GPUs. Poseidon's design is orthogonal to TensorFlow, Caffe or other DL frameworks – the techniques present in Poseidon could be used to produce a better distributed version of them. We empirically show that Poseidon constantly delivers linear speedups using up to 32 nodes and limited bandwidth on a variety of neural network, datasets and computation engines, and compares favorably to Adam and Microsoft CNTK.

Acknowledgments

We thank our shepherd Yu Hua and ATC reviewers for their helpful feedback. We thank the CMU Parallel Data Laboratory for their machine resources and Henggang Cui for insightful discussion. This research is supported by NSF Big Data IIS1447676 and NSF XPS Parallel CCF1629559.

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] CHEN, J., MONGA, R., BENGIO, S., AND JOZEFOWICZ, R. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981* (2016).

[3] 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).

[4] CHILIMBI, T., APACIBLE, Y. S. J., AND KALYANARAMAN, K. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *OSDI* (2014).

[5] COATES, A., HUVAL, B., WANG, T., WU, D. J., NG, A. Y., AND CATANZARO, B. Deep Learning with COTS HPC Systems. In *ICML* (2013).

[6] COLLOBERT, R., KAVUKCUOGLU, K., AND FARABET, C. Torch7: A Matlab-like Environment for Machine Learning. In *NIPSW* (2011).

[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] DAI, W., KUMAR, A., WEI, J., HO, Q., GIBSON, G., AND XING, E. P. Analysis of high-performance distributed ml at scale through parameter server consistency models. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence* (2015).

[9] DEAN, J., CORRADO, G. S., MONGA, R., CHEN, K., DEVIN, M., LE, Q. V., MAO, M. Z., RANZATO, M., SENIOR, A., TUCKER, P., YANG, K., AND NG, A. Y. Large Scale Distributed Deep Networks. In *NIPS* (2012).

[10] DENG, L., LI, J., HUANG, J.-T., YAO, K., YU, D., SEIDE, F., SELTZER, M. L., ZWEIG, G., HE, X., WILLIAMS, J., GONG, Y., AND ACERO, A. Recent Advances in Deep Learning for Speech Research at Microsoft. In *ICASSP* (2013).

[11] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385* (2015).

[12] HO, Q., CIPAR, J., CUI, H., KIM, J. K., LEE, S., GIBBONS, P. B., GIBSON, G. A., GANGER, G. R., AND XING, E. P. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *NIPS* (2013).

[13] IANDOLA, F. N., MOSKEWICZ, M. W., ASHRAF, K., AND KEUTZER, K. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 2592–2600.

[14] JIA, Y., SHELHAMER, E., DONAHUE, J., KARAYEV, S., LONG, J., GIRSHICK, R., GUADARRAMA, S., AND DARRELL, T. Caffe: Convolutional Architecture for Fast Feature Embedding. In *MM* (2014).

[15] KRIZHEVSKY, A. Learning Multiple Layers of Features from Tiny Images. Master's thesis, University of Toronto, 2009.

[16] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS* (2012).

[17] LE, Q. V., MONGA, R., DEVIN, M., CHEN, K., CORRADO, G. S., DEAN, J., AND NG, A. Y. Building High-level Features Using Large Scale Unsupervised Learning. In *ICML* (2012).

[18] LI, H., KADAV, A., KRUUS, E., AND UNGUREANU, C. Malt: distributed data-parallelism for existing ml applications. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM, p. 3.

[19] 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)* (2014), pp. 583–598.

[20] 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).

[21] MIKOLOV, T., CHEN, K., CORRADO, G., AND DEAN, J. Efficient Estimation of Word Representations in Vector Space. In *ICLRW* (2013).

[22] MORITZ, P., NISHIHARA, R., STOICA, I., AND JORDAN, M. I. Sparknet: Training deep networks in spark. *arXiv preprint arXiv:1511.06051* (2015).

[23] 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. *IJCV* (2015), 1–42.

[24] SEIDE, F., FU, H., DROPPO, J., LI, G., AND YU, D. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *INTERSPEECH* (2014), pp. 1058–1062.

[25] SEIDE, F., FU, H., DROPPO, J., LI, G., AND YU, D. On parallelizability of stochastic gradient descent for speech dnns. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2014), IEEE, pp. 235–239.

[26] SIMONYAN, K., AND ZISSERMAN, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR* (2015).

[27] SZEGEDY, C., LIU, W., JIA, Y., SERMANET, P., REED, S., ANGUELOV, D., ERHAN, D., VANHOUCKE, V., AND RABINOVICH, A. Going deeper with convolutions. In *CVPR* (2015).

[28] SZEGEDY, C., VANHOUCKE, V., IOFFE, S., SHLENS, J., AND WOJNA, Z. Rethinking the inception architecture for computer vision. *arXiv preprint arXiv:1512.00567* (2015).

[29] WANG, W., CHEN, G., DINH, T. T. A., GAO, J., OOI, B. C., TAN, K.-L., AND WANG, S. SINGA: Putting Deep Learning in the Hands of Multimedia Users. In *MM* (2015).

[30] WATCHARAPICHAT, P., MORALES, V. L., FERNANDEZ, R. C., AND PIETZUCH, P. Ako: Decentralised deep learning with partial gradient exchange. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (2016), ACM, pp. 84–97.

[31] 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 *SoCC* (2015).

[32] XIE, P., KIM, J. K., ZHOU, Y., HO, Q., KUMAR, A., YU, Y., AND XING, E. Distributed Machine Learning via Sufficient Factor Broadcasting. In *arXiv* (2015).

[33] YAHOO. Caffe on spark. `http://yahoohadoop.tumblr.com/post/129872361846/large-scale-distributed-deep-learning-on-hadoop`.

[34] 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).

[35] 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.

[36] YU, D., EVERSOLE, A., SELTZER, M., YAO, K., HUANG, Z., GUENTER, B., KUCHAIEV, O., ZHANG, Y., SEIDE, F., WANG, H., ET AL. An introduction to computational networks and the computational network toolkit. Tech. rep.

[37] 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).

[38] ZOU, Y., JIN, X., LI, Y., GUO, Z., WANG, E., AND XIAO, B. Mariana: Tencent Deep Learning Platform and its Applications. In *VLDB Endowment* (2014).

Garaph: Efficient *G*PU-*a*ccelerated G*raph* Processing on a Single Machine with Balanced Replication

Lingxiao Ma[§#], Zhi Yang[§#*], Han Chen[§], Jilong Xue[†] and Yafei Dai[‡]
[§]*Computer Science Department, Peking University, Beijing, China*
[†]*Microsoft Research, Beijing, China*
[‡]*Institute of Big Data Technologies Shenzhen Key Lab for Cloud Computing Technology & Applications, Peking University, China*

Abstract

Recent advances in storage (e.g., DDR4, SSD, NVM) and accelerators (e.g., GPU, Xeon-Phi, FPGA) provide the opportunity to efficiently process large-scale graphs on a single machine. In this paper, we present Garaph, a GPU-accelerated graph processing system on a single machine with secondary storage as memory extension. Garaph is novel in three ways. First, Garaph proposes a vertex replication degree customization scheme that maximizes the GPU utilization given vertices' degrees and space constraints. Second, Garaph adopts a balanced edge-based partition ensuring work balance over CPU threads, and also a hybrid of notify-pull and pull computation models optimized for fast graph processing on the CPU. Third, Garaph uses a dynamic workload assignment scheme which takes into account both characteristics of processing elements and graph algorithms. Our evaluation with six widely used graph applications on seven real-world graphs shows that Garaph significantly outperforms existing state-of-art CPU-based and GPU-based graph processing systems, getting up to 5.36x speedup over the fastest among them.

1 Introduction

Triggered by the availability of graph-structured data in domains ranging from social networks to genomics and business, the need for efficient large scale graph processing has grown. The resulting demand has driven the development of many distributed systems, including Pregel [22], Giraph [2], GraphX [11], GraphLab [21], PowerGraph [10], PowerLyra [7] and Gemini [36]. These systems attempt to scale to graphs of billions of edges by distributing the computation over multiple cluster nodes. However, the performance of existing graph frameworks relies on effective partitioning to minimize communication, which is very difficult for natural graphs [1, 20, 10]. Therefore, network performance required for communication between graph partitions emerges as the bottleneck, and thus distributed graph systems require very fast networks to realize good performance.

As an alternative, several non-distributed graph processing systems have been proposed. Galois [27] and Ligra [32] are specific for shared-memory/multi-core machines, whereas GraphChi [17], X-Stream [29] and GridGraph [37] are designed for processing large graphs on a single machine, by relying on secondary storage. Such solutions no longer require the resources of very large clusters, and users need not to be skilled at managing and tuning a distributed system in a cluster.

But the large amount of data to be processed in a single machine put pressure on two scarce resources: memory and computing power. We observe, however, that today more efficient non-distributed solutions are affordable. On the one hand, current commodity single unit servers can easily aggregate hundreds of GBs to TBs of RAM [32]. Further, with recent advances of secondary storage such as SATA/PCIe-based solid-state drive (SSD) and non-volatile memory (NVM), it is feasible to aggregate multiple secondary storages to achieve a high access bandwidth close to memory. On the other hand, current GPUs have much higher massive parallelism and memory access bandwidth than traditional CPUs, which has the potential to offer high-performance graph processing.

Given these recent advances, GPU-accelerated, secondary-storage based graph processing has the potential to offer a viable solution. However, while several attempts [8, 34] have been made recently, efficient large-scale graph computation on CPU/GPU hybrid platforms still remains a challenge due to the highly skewed degree distribution of the natural graphs and heterogeneous parallelism of CPU and GPU. In particular, the skewed degree distribution implies that a small fraction of the vertices are adjacent to a large fraction of the edges. This

*Corresponding author (yangzhi@pku.edu.cn)
#These authors contributed equally to this work.

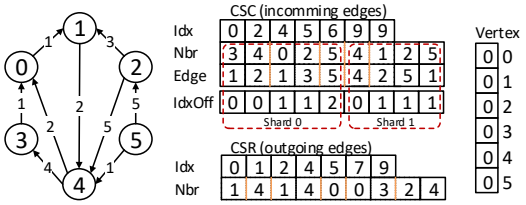


Figure 1: Graph Representation in Garaph

concentration of edges results in heavy write contention among GPU threads due to atomic updates of the same vertices. Colliding threads will be serialized, seriously harming performance on GPUs. Further, the power-law degree distribution can lead to substantial work imbalance across CPU threads in existing non-distributed graph systems that treat vertices symmetrically. Thus, its impact on the parallelism of both GPU and CPU should be alleviated by effective optimization techniques. Another challenge is how to efficiently integrate heterogeneous parallelism of CPU and GPU for graph processing under an unified abstraction, as CPUs are fast for the sequential processing whereas GPUs are suitable for the bulk parallel processing.

In this context, we present Garaph, a non-distributed system that supports *GPU-accelerated* graph processing with secondary storage as memory extension. Garaph enables using all CPU and GPU cores on a given node for graph processing, and with Garaph's abstractions, users only need to write one program that can be executed by both CPU and GPU. Besides, Garaph uses an array of SSDs to achieve high throughput and low latency storage, which enables the system to process large-scale graphs exceeding the computer's memory capacity.

Garaph is novel in the following aspects: To cope with the heterogeneity of vertex degree, Garaph proposes a vertex replication degree customization scheme on the GPU side that maximizes the GPU utilization given vertex degree and space constraints. On the CPU side, Garaph adopts a balanced edge-based partition to ensure work balance over CPU threads. For the heterogeneity of computation units, Garaph first uses a pull computation model matching the SIMD processing model of GPU, and a hybrid of notify-pull and pull computation models optimized for fast sequential processing on the CPU. Further, Garaph uses a dynamic workload assignment scheme which takes into account both the characteristics of processing elements and the properties of graph applications. These new schemes together make for an efficient implementation, achieving full parallelism on both CPU and GPU sides in a single machine.

We evaluate our Garaph prototype with extensive experiments and compared it with four state-of-the-art systems. Experiments with six applications on seven real-world graphs demonstrate that Garaph significantly outperforms existing state-of-art CPU-based and GPU-based systems, getting a speedup of 2.56x on average (up to 5.36x). Through solving conflicts in computation, customized replication scheme can improve GPU's performance by 4.84x speedup on average (up to 32.15x).

2 System Overview

In this section, we give a brief overview on the graph representation, the architecture and the computation abstraction of Garaph.

2.1 Graph Representation

Garaph adopts both *Compressed Sparse Column (CSC)* and *Compressed Sparse Row (CSR)* for organizing incoming and outgoing edges, respectively. The index array `Idx` records each vertex's edge distribution: for vertex i, `Idx[i]` and `Idx[i+1]` indicate the beginning and ending offsets of its incoming/outgoing edges. The array `Nbr` records sources of incoming edges or destinations of outgoing ones. The arrays `Vertex` and `Edge` record values of vertices and edges, respectively. For example, as shown in CSC of Figure 1, `Idx[0]` and `Idx[1]` indicates that vertex 0 has two incomming edges with sources $3,4$ and values $1,2$, respectively.

As stated in CuSha [16], the CSR representation is not friendly for processing graphs in GPUs, which could incur high frequency of non-coalesced memory accesses and warp divergence. To overcome this problem, we also adopt the concept of **shard**. In particular, we split the vertices V of graph $G = (V,E)$ into disjoint sets of vertices and each set is represented by a shard that stores all the incoming edges whose destination is in that set. Edges in a shard are listed based on increasing order of their indexes of destination vertices. Given sorted edges, we index the destination of each edge by the offset to the first destination vertex in this shard, represented by a array of `IdxOff`, for example, as illustrated in Figure 1, edges with destination $3,4,5$ are in shard 1 and destinations' `IdxOff` are $0,1,2$, respectively.

To improve GPU utilization, we allow each shard to be fit into the shared memory for high bandwidth. Specifically, the number of vertices in each shard is determined by $C_{shm}/(N_{Block} \cdot S_{vertex})$, where C_{shm} is the size of the shared memory, N_{Block} is the number of threads blocks in the SM and S_{vertex} is the size of one vertex. However, if a chunk adopts vertex replication with a factor of R (described later), the shard size should reduce by R times. In practice, one block of GPU threads can use up to 48KB shared memory. Let $S_{vertex} \geq 32$ bits for billion-scale graphs, so each shard contains at most 12K vertices. This also implies that maximum offset in `IdxOff` is 12K, so we can use 16-bit integer to represent the index of destination vertices. This compression could not only save GPU memory, but also reduce the traffic of copying

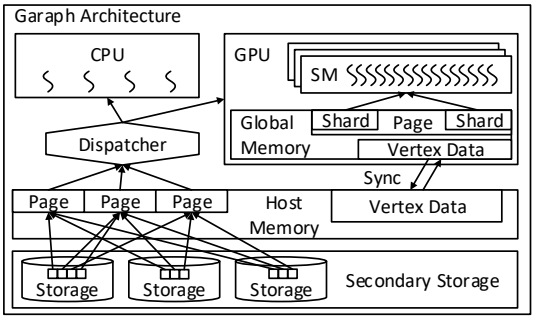


Figure 2: Garaph Architecture

a shard from the host memory to the GPU memory.

To improve the efficiency of CPU-GPU memory copy, we transfer the shards from host memory to GPU memory in batch. In the following, we call the set of shards transferred in a batch as a **page**. Each page contains the maximum number of consecutive shards that can be stored completely in the GPU memory. As we shall describe later, the system also leverages the multi-stream feature of GPUs for the overlap of memory copy and kernel execution. Let N_s be the number of streams. In this case, the page size is chosen as the maximum number of shards that can be stored in $1/N_s$ of the GPU memory.

With above graph representations, our system adopts the following two vertex-centric computation models. The first is *pull* model where every vertex updates its state by pulling the new states of neighboring vertices through incoming edges. The other is the *notify-pull* model where only the active vertices notify their outgoing neighbors to update, who in turn perform local computation by pulling states of their incoming neighbors. Clearly, this model is more effective in case of few active vertices. Note that the CSR is only used for notification. To save memory usage, Garaph does not store the values of outgoing edges in the CSR (see Figure 1).

2.2 System Architecture.

Figure 2 shows the architecture of Garaph, which consists of three main components: dispatcher, CPU and GPU computation kernels.

Dispatcher. This functional module is responsible for loading graph from secondary storages, distributing the computation over CPU and GPU and making adjustment if necessary. To exploit I/O parallelism, Garaph partitions the each graph page of into equal-size data blocks, which are uniformly distributed over multiple secondary storages (e.g., SSDs) with a hash function.

To process the graph, data blocks are loaded from the secondary storages to the host memory to construct pages by the dispatcher. After one page has been constructed, it will be dispatched to either the CPU or the GPU.

GPU/CPU computation kernel. These two kernels

```
interface GASVertexProgram(u)
  gather(D_u, D_(u,v), D_v) → Accum
  sum(Accum left, Accum right) → Accum
  apply(D_u, Accum) → D_u^new
  activate(D_u^new, D_u) → A[u]
```

Figure 3: Garaph API

are in charge of graph processing. After receiving a page from the dispatcher, the GPU kernel processes the shards of page in a parallel manner, where each shard is processed by a block in the GPU. For efficient graph processing, only the pull model is enabled on the GPU side. This is because the notify-pull model can lead to high frequency of non-coalesced memory accesses because of poor locality and warp divergence caused by distinguishing active/inactive vertices, significantly limit its performance while processing graphs in the GPU.

The CPU kernel enables both *pull* and *notify-pull* computation models. To balance the computation across multiple threads, the kernel divides edges of a page into sets of equal size, with each thread processing one edge set. When either of two kernels has processed one page, there will be a synchronization between the CPU and the GPU. We shall describe these two kernels in Section 3 and 4.

Garaph enables programs to be executed both synchronously and asynchronously. As the system processes the graph page-by-page, we define an *iteration* as a complete process over all the pages for one time, irrespective of synchronous or asynchronous execution. The synchronous execution model ensures a deterministic execution regardless of the number of machines and closely resembles Pregel [22]. Changes made to the vertex data are committed at the end of each iteration and are visible in the subsequent iteration. When run asynchronously, changes made to the vertex data are immediately committed and visible to current and subsequent iterations. The system terminates the processing if the graph state converges or a given number of iterations are completed.

Fault Tolerance. Garaph enables fault tolerance by writing the vertex data to secondary storages periodically. When Garaph runs synchroniously, Garaph will write the vertex data into stable storages after one or several iterations (user-defined). When Garaph runs asynchroniously, Garaph will write the updated vertex data from the main memory into stable storages after one page has been processed or write the whole vertex data into stable storages after one or several iterations (user-defined). When a fault occurs, it loads the vertex data from the secondary storage and continues to run the application.

2.3 Programming APIs

Garaph implements a modified Gather-Apply-Scatter (GAS) abstraction used in PowerGraph [10]. For a vertex

```
template<typename T>
__host__ __device__ void unifiedAdd(T *addr, T val) {
#ifdef __CUDA_ARCH__ // For GPU, atomic operation
    atomicAdd(addr, val);
#else // For CPU, non-atomic operation
    *addr += val;
#endif
}
```

Figure 4: Garaph unifiedAdd Operation

u, the *gather* function is passed the data on the adjacent vertex D_v and edge $D_{(u,v)}$ and returns a temporary accumulator *Accum*, which is combined using the commutative and associative *sum* operation. When algorithms are commutative and associative, this abstraction is equivalent to original GAS abstraction [10]. After the gather phase has completed, the *apply* function takes the final accumulator and computes a new vertex value D_u^{new}.

In the GAS abstraction, the scatter function is invoked to produce new edge value $D_{(u,v)}^{new}$ which are written back to different machines in a distributed environment. However, Garaph is specific for a non-distributed platform, where each vertex can access its neighbors' updated values in memory so that pushing new data over edges is unnecessary. Thus, we modify the *scatter* function to *activate* function which sets $A[u] = 1$ if vertex u satisfies the active condition defined in the function (e.g., there is a significant change in the vertex value), otherwise, the function sets $A[u] = 0$ to indicate u is inactive. Figure 3 shows the functions Garaph supports.

Garaph packages a set of operations with atomic and non-atomic as unified operations (e.g. unifiedAdd in figure 4) which covers all the atomic operations of CUDA [25]. In Garaph, users only need to write one kernel code which could be executed by both CPU and GPU. Since multiple GPU threads might simultaneously modify the same memory address, the user-provided sum function must be atomic on the GPU side. Garaph packages the non-atomic operations for CPU and the atomic operations for GPU into one set of operations, so that users could directly invoke irrespective of implementing the program on GPU or CPU.

3 GPU-Based Graph Processing

In this section, we first describe how Garaph executes iterative parallel graph algorithms on the GPU side. We then propose an vertex replication degree customization scheme that maximizes the expected GPU performance given properties of the graph and the GPU.

3.1 Graph Processing Engine

To facilitate efficient processing of graphs on GPU using shards, the GPU kernel maintains an array named

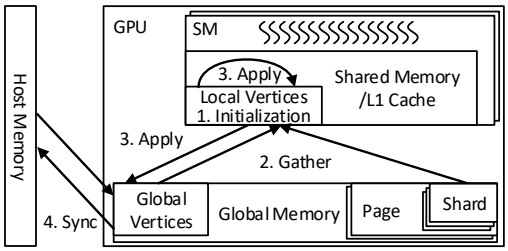


Figure 5: GPU-Based Graph Processing Engine

GlobalVertices in the global memory, which allows quick access to the values of vertices. Current GPUs can support up to 24GB global memory, whereas the size of vertices is usually 4 bytes (FP32 or INT) or 2 bytes (FP16). So GPUs can store up to 6 billion (or 12 billion) vertices in global memory, which is sufficient for most datasets, for example, the largest open source dataset (HyperLink12 [18]) has 3.5 billion vertices.

Multiple shards of a page are processed by threads blocks running on many streaming-multiprocessors (SM) in a parallel manner, where each shard is processed by a thread block. As illustrated in Figure 5, each shard in a page is processed by one GPU block in three phases: *initialization*, *gather* and *apply*. When the page has been processed, the new vertex values are *synchronized* between GPU global memory and host memory. Let S_i represent the destination vertices in one shard.

Initialization. At the beginning, the GPU allocates an array *LocalVertices* in the shared memory of this SM to store the accumulate value of each vertex in a shard. Then, consecutive threads of a block initialize this array with default vertex values defined by users, e.g., 0 for the PageRank application.

Gather. Threads of one GPU block process edges of an individual shard. For each edge (u,v), one thread fetches vertex and edge data from the global memory and increases the accumulate value: $a_u \leftarrow \text{sum}(a_u, \text{gather}(D_u, D_{(u,v)}, D_v)), \forall v \in \mathbf{Nbr}[u]$. To have coalesced global memory accesses, consecutive threads of the block read consecutive edges' data in global memory.

Apply. Each thread of a block updates the vertex value in the shared memory: $D_u^{new} \leftarrow \text{apply}(D_u, a_u), \forall u \in S_i$. Consecutive threads of this block process consecutive vertices. When executing programs asynchronously, the system commits new vertex data to the *GlobalVertices* array, which are immediately visible to the subsequent computation. Otherwise, these values are written to a temporary array in the global memory, which would be visible in the next iteration.

Synchronization from GPU to CPU. Once the whole graph page has been processed, the GPU global memory will be synchronized with the host memory. For programs executed asynchronously, the system transmits the

updated values of the *GlobalVertices* in the GPU global memory to the array storing the most updated values of vertices in the host memory. For those executed synchronously, updated values stored in the temporary space of the GPU global memory are transmitted to a temporary array in the host memory, which will be committed after this iteration ends. As the PCIe bus is full-duplex and most GPUs have two copy engines, the synchronization can be overlapped with processing pages in GPU.

3.2 Replication-Based Gather

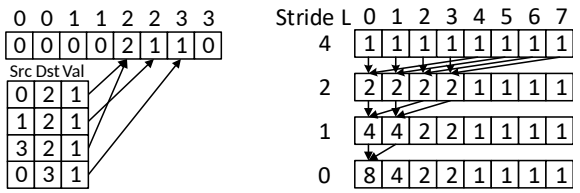
Our above GPU-accelerated framework provides a convenient environment to write graph processing applications. However, the current design still suffers the write contention problem in the gather phase, since multiple threads might collide while simultaneously modifying the same shared memory address (e.g., processing edges with the same destination). Such a collision is called as *position conflict*. The position conflict typically entails a need to serialize memory updates that is resolved by using atomic operations. These consist of a memory read, an arithmetic operation, and a memory write, entailing a latency penalty that is proportional to the number of colliding threads n: a n-way position conflict incurs a penalty of $(n-1)\times t_{position}$, where $t_{position}$ is the processing time of one atomic operation [9].

Notice that natural graphs in the real-world have highly skewed power-law degree distributions, which implies that position conflicts will be very frequent, especially for those vertices of high degree. The heavy write contention leads to an immense performance bottleneck on the GPU side, so its impact on the gather phase should be alleviated by effective optimization techniques.

A general strategy for reducing the conflicts is *replication*, which consists of placing R adjoining copies of the partial accumulated value a'_u in the shared memory to spread these accesses over more shared memory addresses. Then these R partial accumulated values are aggregated to calculate the final accumulated value a_u for a vertex u. Here, R is called as *replication factor*.

Mapping and Aggregation To implement the replication, a mapping function is needed to assign to each thread a replicated copy of the vertex, where the thread will perform the atomic operations. For efficient mapping, we require the vertices of a shard S_i have the same replica factor of R_i. So for any vertex u_i in this shard, the mapping function used in our system is given by: $addr[u'_i] = (i - r_s) * R_i + tid\%R_i, \forall u_i \in S_i$, where $addr[u'_i]$ represents the address of the replica u'_i assigned to the thread tid, and the r_s is the beginning index of the shard.

The mapping makes consecutive threads access consecutive copies. With the mapping, threads perform gather on multiple replicas in a parallel manner: $a'_u \leftarrow \text{sum}(a'_u, \text{gather}(D_u, D_{(u,v)}, D_v)), \forall v \in \mathbf{Nbr}[u']$, where u' is a replica of vertex u.



(a) Mapping ($R = 2$) (b) Aggregation ($R = 8$)

Figure 6: Mapping and Aggregation

After all the edges in the shard have been processed, the system needs a additional phase to aggregate values of different replicas in the shared memory, i.e., $a_u \leftarrow \text{sum}(a_u, a'_u)$. For fast aggregation, we set $R_i = 2^n (n \geq 0)$ to implement a two-way merge illustrated in Figure 6(b). In each iteration, the thread tid executes the user-defined *sum* function with a stride length of L. Initially, $L = R_i/2$, after all replicas are processed, $L \leftarrow L/2$ and the next iteration begins. This procedure stops until $L < 1$.

Replication Factor Customization. Although replication can reduce write conflicts, excessive replication could still lead to GPU underutilization in that fewer vertices can be fit in the shared memory. To achieve a balanced replication, we propose a replication factor customization scheme that maximizes the expected performance under given conflict degree and space constraints.

To do so, we model the execution time of the replication-based gather phase to examine impact of R_i. Let V_i, E_i be the number of destination vertices and edges in a shard S_i, and let t_l and t_a be the time of accessing the global memory and executing an atomic operation in the shared memory, respectively.

In the gather phase, to process a shard S_i with replication factor R_i, threads read $R_i \times |V_i|$ vertices' data and $|E_i|$ edges' data from the global memory into the shared memory, thus taking $R_i \times |V_i| \times t_l$ and $|E_i| \times t_l$, respectively. In the shared memory, threads execute $|E_i|$ atomic operations with the average conflict degree of $\frac{|E_i|}{|V_i| \times R_i}$, which takes $\frac{|E_i|^2}{|V_i| \times R_i} \times t_a$. The final step of aggregation takes $|V_i| \times t_a \times \log R_i$. Thus, the total time $T_G(R_i)$ of processing the shard S_i is given by:

$$T_G(R_i) = R_i|V_i|t_l + |E_i|t_l + \frac{|E_i|^2}{|V_i|R_i}t_a + |V_i|t_a \log R_i. \quad (1)$$

Our goal is to find the R_i minimizing $T_G(R_i)$. We simplify the above equation with $\log R_i \ll R_i$, and $T(R_i)$ is minimized when $R_i \times |V_i| \times t_l = \frac{|E_i|^2}{|V_i| \times R_i} \times t_a$.

Solving the above equation, we get the best replication factor for a shard S_i as: $R_i = \frac{|E_i|}{|V_i|} \cdot \frac{t_a}{t_l}$. In practice, $t_a \approx t_l$, and we get $R_i = \frac{|E_i|}{|V_i|}$. Notice that position conflicts between two consecutive threads will be complete-

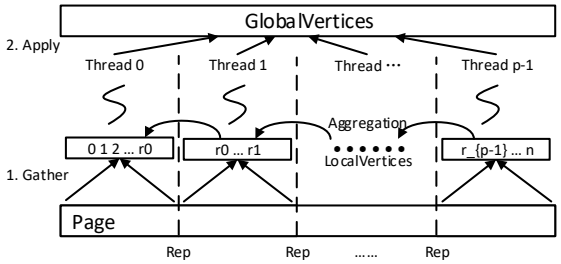


Figure 7: Processing with Edge Partitions on the CPU

ly removed when the replication factor is up to 32, i.e., one copy for each thread of one warp [13]. Also recall $R_i = 2^n (n \geq 0)$ for efficient aggregation, so we choose the value of 2^n form that is closest to average degree of a shard as its final replication factor:

$$R_i = 2^{\min\left\{\left\lceil \log \frac{|E_i|}{|V_i|} - 0.5 \right\rfloor, 5\right\}}. \quad (2)$$

Our replication policy suggests that the number of replicas for each vertex should be customized to the average degree of vertices in a given shard. Specifically, shards containing more high-degree vertices tend to have higher replication degrees for a larger probability of reduction in conflicts.

4 CPU-Based Graph Processing

In this section, we first describe the graph processing of Garaph on the CPU side, which adopts a balanced edge-based partition to exploit full parallelism. We then describe the dual-mode processing model of Garaph, which adaptively switches between the pull/notify-pull modes according to the density of active vertices in the page.

4.1 Processing with Edge Partitions

Existing single-node graph systems treat vertices symmetrically and parallelize the graph processing by assigning each thread a subset of vertices to process. However, this method leads to substantial computation imbalance due to the power-law degree distribution. Further, it also increases the random memory access of edge data if adjacent vertices are assigned to different threads. These issues degrade the overall system performance.

Different from common systems, Garaph adopts edge-centric partition. As illustrated in Figure 7, the edges of a page are equally partitioned across threads, where multiple CPU threads process independent edge sets in a parallel manner. Vertices cut at the partition boundaries would be replicated, and the system would aggregate the replicas' values to obtain the vertex value. This partition enhances the sequential access of edge data and improves work balance over threads.

The CPU engine also maintains a *GlobalVertices* array in the host memory for quick access to values of vertices. Each page is processed in three stage: *initialization*, *gather*, *apply*. If a page has been processed on the GPU side, the system also synchronizes new vertex values between the GPU memory and the host memory. For a common graph application, the processing is done by pulling new vertex states along outgoing edges, until the graph state converges (e.g., no active vertices) or a given number of iterations are completed. Vertices with significant state change are called active vertices (determined by *activate*() function). We use a bitmap A to indicates the inactive/active state of each vertex.

Initialization. Let n_t be the number of CPU threads. The edges of the page is divided to n_t partitions of the same size, and thread tid processes the tid_{th} partition. The fist and last vertices of partition will create a replica respectively if they are cut at the boundaries. So the number of replicas is at most $n_t - 1$. Each CPU thread maintains a *LocalVertices* array to store the accumulate values of destination vertices in the corresponding partition. Like the GPU, this array is initialized with the vertices' default value defined by users.

Gather. Each partition of the page is processed by one CPU thread. For each edge, the CPU thread performs gather and updates the accumulate value a_u in *LocalVertices* with sum function. Edges are processed in a sequential order whereas the source vertices' values are accessed randomly by each thread. After each thread has processed its partition, an *aggregation* phase aggregates values of vertices replicated at the partition boundaries. Recall that the number of replicas is at most $n_t - 1$, which is small enough for one CPU thread to process. In Garaph, thread 0 scans the whole partitions in the reverse order and aggregates values of replicated vertices, as illustrated in Figure 7.

Apply. After the gather phase of each page is finished, every thread updates vertices' values in their own *LocalVertices* array. For each partition, if the first vertex is replicated, it will be ignored because its value has already been aggregated to the last vertex of the previous adjacent partition. For each vertex in a partition, the corresponding thread calls *activate*() function to examine if the vertex is active or not and updates the bitmap A.

Synchronization from CPU to GPU. As described in Section 3.1, after the GPU has processed a page, it sends the corresponding vertex values to the host memory. When receiving the new values, the system first calls *Activate*() function to update the bitmap A of these updated vertices. When runs asynchronously, the system enables the updates received from the GPU immediately visible through writing them into the *GlobalVertices* array of host memory. After that, the system sends back new ver-

tices updated on CPU side to the GPU and overwrites the corresponding part of the *GlobalVertices* array in the GPU global memory. When run synchronously, the system stores updated values in a temporary array, and commits these new values at the end of each iteration. In this case, the CPU transmits the new *GlobalVertices* array to that in the GPU memory at the end of each iteration.

4.2 Dual-Mode Processing Engine

To this end, our CPU-based processing engine adopts a *pull* mode where every vertex performs local computation by pulling the states of neighboring vertices through incoming edges. However, a vertex needs to be updated only when one of its source vertices is active in the previous iteration. Thus, another way to process vertices is *notify-pull* mode where only the vertices activated in the last iteration notify their outgoing neighbors, who in turn perform local computation by pulling states of their incoming neighbors. Intuitively, the notify-pull mode is more efficient when few vertices are active in the last iteration (sparse active vertex set), as the system only traverses the outgoing edges of active vertices where new updates to be made. In contrast, the pull mode is more beneficial when most vertices are activated (dense active vertex set), which avoids the extra cost of notifications.

At a given time during graph processing, the active vertex set may be dense or sparse. For example, SSSP or the BFS starts from a very sparse set, becoming denser as more vertices being activated by their in-neighbors, and sparse again when the algorithm approaches convergence. To incorporate the benefits of both modes, we extend our CPU processing to a dual engine design determined by the size of the vertex set $V_A = \{v|(u,v) \in E, A[u] = true\}$ given graph $G(V,E)$, i.e., the outgoing neighbors of vertices activated in the last iteration.

We first consider the case where the graph representations can be completely loaded into the host memory. Let T_{pull} be the time of graph processing in the pull mode. Clearly, T_{pull} is independent of $|V_A|$. In contrast, the notify-pull mode only notifies a fraction of $f = |V_A|/|V|$, who are updated in turn. Hence, we estimate the average processing time in this mode as $T_{notify-pull} = 2fT_{pull}$, as the time to notify is at most equal to pulling state along edges. So the system would adopt *notify-pull* mode if the $f \leq 1/2$ (i.e., $T_{notify-pull} < T_{pull}$), otherwise, the *pull* model would be adopted.

However, for the scenario where only part of graph can be loaded into the host memory, the system entails I/O cost due to sequential and random accesses of outgoing/incoming edges on secondary storage for pull and notify-pull modes respectively. Let k $(k > 1)$ be the rate of speeds between sequential read and random read (e.g., $k \approx 10$ in SSD), the $T_{notify-pull} = 2kf \cdot T_{pull}$. In this case, the system would adopt *notify-pull* mode if $f \leq \frac{1}{2k}$.

Let $\Gamma = \{u|A[u] = 1\}$ be the set of active vertices in the last iteration, the system can estimate $f \approx \frac{\sum_{u \in \Gamma} d_u}{|E|}$ where d_u is the out-degree of an active vertex u and E is the set of edges. Garaph estimates f in the beginning of each iteration and choose which mode to use based on f.

5 Dispatcher

We have discussed how to design and optimize graph processing kernel for efficient execution on both GPU and CPU sides. To further improve the performance, we propose an adaptive scheduling mechanism to exploit the overlap of two engines. Besides, we also perform multi-stream scheduling for data transfer and GPU kernel execution overlap. Below we shall detail each scheduling strategy respectively.

5.1 CPU-GPU Scheduling

We first determine when it is beneficial to adopt GPU acceleration. From Section 4.2 we know that the processing time with only CPU kernel is $T_{CPU} = \min\{2f\rho, 1\} \cdot T_{pull}$, where f is the fraction of vertices to be updated and T_{pull} is processing time in the pull mode. Here, $\rho = 1$ if the graph representations can be fit into the host memory, otherwise, $\rho = k$, which is the rate of speeds between sequential read and random read of secondary storage. Notice that GPU-based kernel needs to process all the edges of a given page, so the GPU's processing time T_{GPU} is independent of f. Therefore, if $T_{CPU} < T_{GPU}$, the system adopts CPU kernel only due to sparse active vertex set. Otherwise, Garaph adopts both GPU and CPU kernels to reduce the overall time of the processing.

Based on the above insight, our scheduler works as follows: At beginning of every iteration, the scheduler calculates the following ratio of T_{CPU} to T_{GPU}:

$$\alpha = \min\{2f\rho, 1\} \cdot \frac{T_{pull}}{T_{GPU}}, \tag{3}$$

where the fraction T_{pull}/T_{GPU} is initialized by the speed ratio of CPU/GPU hardwares, and is updated once both kernels have begun to process pages. Specifically, let t^p_{cpu} and t^p_{gpu} be the measured time to process a page via CPU and GPU kernels, respectively, we can estimate $T_{pull}/T_{GPU} = t^p_{cpu}/(f \cdot t^p_{gpu})$.

In the case of $\alpha < 1$, only CPU kernel is used for graph processing in this iteration as most vertices are inactive (e.g., a very small f). Otherwise, the system processes graph pages in parallel on both CPU and GPU kernels. In the hybrid mode, the system reactively assigns a page to a (GPU or CPU) kernel once the kernel becomes free. The processing is finished if the graph state converges (i.e., $f = 0$) or a given number of iterations are completed.

Graph	$\|V\|$	$\|E\|$	Max in-deg	Avg deg	Size edgelist
uk-2007@1M	1M	41M	0.4M	41	0.6GB
uk-2014-host	4.8M	51M	0.7M	11	0.8GB
enwiki-2013	4.2M	0.1B	0.4M	24	1.7GB
gsh-2015-tpd	31M	0.6B	2.2M	20	10GB
twitter-2010	42M	1.5B	0.8M	35	27GB
sk-2005	51M	1.9B	8.6M	39	35GB
renren-2010	58M	2.8B	0.3M	48	44GB
uk-union	134M	5.5B	6.4M	41	0.1TB
gsh-2015	988M	34B	59M	34	0.7TB

Table 1: Graph datasets [19, 5, 4, 23] used in evaluation.

5.2 GPU Multi-Stream Scheduling

To trigger the graph processing on the GPU side, there are two threads running on the host: the transmission thread and the computation thread. The former thread continuously transmits each page from the host memory to GPU's global memory. The later thread launches a new GPU kernel to process the page that has already been transmitted.

Using NVIDIA's Hyper-Q feature [24], we perform multi-stream scheduling for the pipelining of CPU-GPU memory copy and kernel execution, so that the processing tasks of pages can be dispatched onto multiple streams and handled concurrently. In particular, we schedule tasks of processing pages onto N_{stream} streams, the transmission thread periodically examines which stream is *idle*, and dispatches the transmission task of one page to the idle stream, where pages are asynchronously transferred from the host memory to the GPU global memory. The computation thread periodically examines which page has been completely transferred, and triggers the computation of that page by dispatching the computation task to the corresponding stream. This multi-stream scheduling enables a high overlapping between CPU-GPU memory copy and kernel execution.

6 Evaluation

In this section, we describe and evaluate our implementation of the Garaph system. Garaph is implemented in more than 8,000 lines of C++ and CUDA code, compiled by GCC 4.8 and CUDA 8.0 on Ubuntu 14.04 with O3 code optimization. In the CPU processing kernel, the number of processing threads is equal to the number of CPU cores by default. In the dispatcher module, each I/O thread reads/writes one secondary storage device so that threads process I/O operations in a parallel manner.

The experiments are performed on a system with Nvidia GeForce GTX 1070 which has 15 SMs (1920 cores) and 8GB global memory. On the host side, there is an Intel Haswell-EP Xeon E5-2650 v3 CPU with 10 cores (hyper-threading enabled) operating at 2.3 GHz clock frequency, and 64GB dual-channel memory. PCI Express 3.0 lanes operating at 16x speed transfer data between the host DDR4 RAM (CPU side) and the device RAM (GPU side).

We use the real-world graphs in Table 1 for evaluation. The largest graph is the gsh-2015 graph with about 1 billion vertices and 34 billion edges. We use six representative graph analytics applications: single source shortest paths (SSSP), connected components (CC), PageRank (PR) [26], neural network (NN) [3], heat simulation (HS) [16], circuit simulation (CS) [16]. We run PR, NN, HS, CS for 10 iterations and CC, SSSP till convergence. To get stable performance, the reported runtime is calculated as the average time of 5 runs.

6.1 Comparison with Other Systems

We compare Garaph of hybrid CPU/GPU kernels (marked as Garaph-H in Table 2) with four state-of-the-art systems: shared-memory systems including CuSha [16], Ligra [32] and Gemini [36], and one secondary-storage-based system GraphChi [17]. Here, CuSha is a GPU framework for processing graphs that can be fit in the GPU memory. To show the performance of each kernel, we also give the performance of Garaph with CPU-kernel only and GPU-kernel only (marked as Garaph-C and Garaph-G in Table 2, respectively).

For datasets that can be placed in host memory, Table 2 presents the performance of evaluated systems. Benefit from customized replication for reducing position conflicts, Garaph-G significantly outperforms CuSha in all cases, 2.34x on average and up to 3.38x for PR on the uk-2014-host dataset. Garaph-G also outperforms other CPU-based systems in compute-intensive applications such as PR, NN, HS and CS. But for SSSP and CC, both Garaph-G and CuSha take longer time to get convergence, as they have to process the whole graph despite of a few active vertices to be processed.

With balanced replication and the optimization for sequential memory access, Garaph-C also outperforms existing systems in many cases: e.g., for PR excluding enwiki-2013 and sk-2005 datasets, for NN and SSSP excluding renren-2010 dataset and for CC in all datasets. Adaptive dual-mode processing engine enables Garaph-C to significantly outperform GPU-based systems in the cases of SSSP and CC.

The above results reveal that Garaph-G is suitable for compute-intensive applications whereas and Garaph-C performs well in applications like SSSP and CC. With the CPU-GPU scheduling, Garaph-H combines the advantages of both Garaph-C and Garaph-G. As a result, Garaph-H significantly outperforms other systems in all cases, e.g., 2.56x on average and up to 5.36x for CC on the twitter-2010 dataset.

	PR						NN					
Graph	CuSha	Ligra	Gemini	Garaph-C	Garaph-G	Garaph-H	CuSha	Ligra	Gemini	Garaph-C	Garaph-G	Garaph-H
uk-2007@1M	0.48	0.77	0.43	0.43	0.20	**0.16**	0.70	0.78	0.44	0.33	0.33	**0.20**
uk-2014-host	0.83	1.06	0.74	0.60	0.25	**0.22**	0.82	0.98	0.88	0.46	0.41	**0.26**
enwiki-2013	1.39	1.80	0.96	1.29	0.49	**0.39**	1.63	1.27	1.18	0.96	0.81	**0.52**
gsh-2015-tpd	-	11.80	8.08	7.80	2.91	**2.42**	-	11.00	7.78	5.91	4.83	**3.36**
twitter-2010	-	-	22.82	22.40	7.50	**6.13**	-	-	20.67	18.53	11.80	**8.61**
sk-2005	-	-	15.85	17.18	9.38	**6.83**	-	-	18.42	12.46	15.67	**9.73**
renren-2010	-	-	84.63	79.77	22.54	**20.89**	-	-	67.60	73.50	22.47	**20.63**
	SSSP						CC					
Graph	CuSha	Ligra	Gemini	Garaph-C	Garaph-G	Garaph-H	CuSha	Ligra	Gemini	Garaph-C	Garaph-G	Garaph-H
uk-2007@1M	6.57	0.57	1.42	0.56	3.95	**0.48**	0.50	0.45	0.38	0.20	0.20	**0.14**
uk-2014-host	13.93	0.73	2.19	0.72	5.88	**0.57**	1.20	0.62	0.72	0.24	0.54	**0.17**
enwiki-2013	11.56	0.99	1.73	0.92	5.54	**0.70**	0.97	0.80	0.77	0.45	0.44	**0.28**
gsh-2015-tpd	-	8.36	9.54	6.70	14.51	**4.32**	-	7.47	5.14	2.58	2.62	**1.21**
twitter-2010	-	-	26.97	23.24	42.49	**12.75**	-	-	17.78	8.78	12.04	**3.32**
sk-2005	-	-	61.04	26.82	1335.29	**18.13**	-	-	13.33	6.49	22.53	**4.74**
renren-2010	-	-	54.60	73.00	615.82	**26.79**	-	-	41.80	35.79	205.03	**8.55**
	HS						CS					
Graph	CuSha	Ligra	Gemini	Garaph-C	Garaph-G	Garaph-H	CuSha	Ligra	Gemini	Garaph-C	Garaph-G	Garaph-H
uk-2007@1M	0.80	0.87	0.30	0.39	0.33	**0.23**	0.80	0.87	0.32	0.43	0.33	**0.23**
uk-2014-host	0.98	1.06	0.71	0.55	0.41	**0.29**	0.99	1.06	0.83	0.59	0.41	**0.29**
enwiki-2013	1.95	1.69	0.98	1.14	0.81	**0.55**	1.95	1.44	0.91	1.19	0.81	**0.55**
gsh-2015-tpd	-	13.80	6.15	6.67	4.84	**3.40**	-	11.10	6.30	4.84	7.20	**3.45**
twitter-2010	-	-	18.83	21.17	11.81	**9.11**	-	-	16.10	21.74	11.80	**8.99**
sk-2005	-	-	13.52	14.34	15.66	**9.95**	-	-	13.24	15.33	15.66	**10.62**
renren-2010	-	-	78.10	82.97	22.47	**21.53**	-	-	59.51	83.84	22.47	**21.29**

Table 2: Runtime (in seconds) of six applications in memory. '-' indicates incompletion due to running out of memory.

	PR		CC	
Graph	GraphChi	Garaph	GraphChi	Garaph
In-memory (10 iters for PR, convergence for CC.)				
uk-2007@1M	2.58	0.16	12.47	0.20
uk-2014-host	4.55	0.22	21.17	0.24
enwiki-2013	7.39	0.39	27.46	0.46
gsh-2015-tpd	39.69	2.42	179.27	1.21
twitter-2010	253.45	6.13	618.58	3.32
sk-2005	-	6.83	-	4.74
renren-2010	-	20.89	-	8.55
Secondary Storage (5 iters for PR and CC.)				
uk-union	899	161	2558	157
gsh-2015	11595	2269	13897	1383

Table 3: Runtime (in seconds) of PR and CC in memory and secondary storages (three SATA SSDs). '-' indicates incompletion due to running out of memory.

As uk-union and gsh-2015 datasets can be only placed in secondary storage, CuSha, Ligra and Gemini cannot run any applications due to the limit of memory capacity.

We compare Garaph with GraphChi in two ways: (1) For datasets that can be fit in memory, we redirect GraphChi's I/O operations from secondary storages to memory by modifing its open-source code. We run PR for 10 iterations and CC till convergence. (2) For datasets that need the extension of secondary storage, we run GraphChi on a Raid-0 provided by three SATA SSDs. In this case, Garaph also uses the same SSDs managed by the dispatcher. We only run 5 iterations for PR and CC on large-scale graphs such as uk-union and gsh-2015 that are very time-consuming to get convergence.

Table 3 shows that Garaph outperforms GraphChi in all cases. The experiments of in-memory graphs demonstrate that Garaph's computation engine is more efficient than GraphChi. There are three reasons why Garaph outperforms GraphChi for SSD-based computation: First, benefiting from the the compressed graph representation, Garaph can use less space to store graph data. Second, GraphChi needs both read data from SSDs and write data to SSDs, which may also cause I/O conflicts. Garaph only reads data from SSDs. Futher, in SSDs, the sequential read speed is much faster than the sequential write speed. Finally, according to our tests, Garaph's disk manager is more efficient than the RAID-0 supported by the raid card which is not linear scalable. Note this is a just preliminary result of adopting the secondary storages. We shall try PCIe SSDs or NVMs in the future work.

6.2 Customized Replication

In this section, we evaluate the performance of the customized replication on the GPU side. We partition the sk-2005 dataset into 33 subgraphs (pages) of similar sizes but different topological structures. We run PR on these pages to evaluate the runtime (computation time only) of each page by using the CUDA toolkit profiler.

Figure 8 shows the runtime of each page in one iteration with/without replication. Without replication, the processing time of pages varies significantly, where the slowest one is about 45.17x slower than the fastest one. We also find that the correlation between the runtime and the maximum degree of individual pages is 0.9853, which implies that the time of processing a page is main-

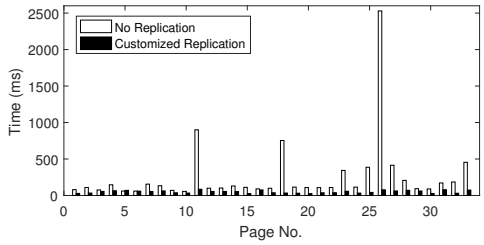


Figure 8: Per-page runtime (computation time only) of PR on the sk-2005 dataset

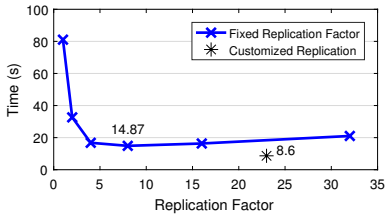
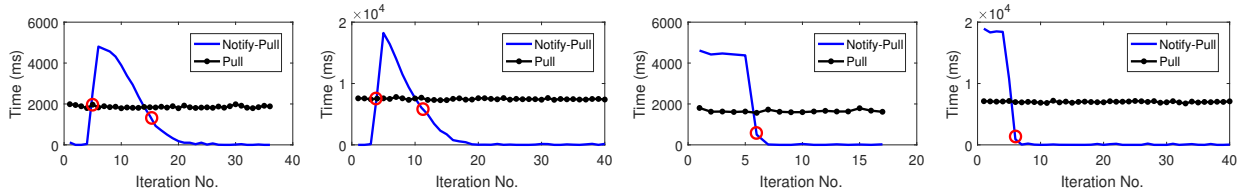


Figure 9: Runtime with different replication factors, the X-axis of customized replication is the average of R_i.



(a) SSSP: twitter-2010 (b) SSSP: renren-2010 (c) CC: twitter-2010 (d) CC: renren-2010

Figure 10: Runtime under notify-pull and pull modes, "o" indicates the iteration where Garaph switches modes.

ly impacted by the vertices of high degree. In contrast, with the customized replication, the processing time of pages is much more balanced and efficient, getting a 4.84x speedup on average (up to 32.15x), significantly reducing the overall time.

We next show customized replication can gain a better performance than a fixed replication factor. To do so, we run 10 iterations PR on the sk-2005 dataset with the fixed factor $R \in \{1,2,4,8,16,32\}$ for the whole graph. Figure 9 shows that the runtime (computation time only) decreases at the beginning and increases with the growing of R. Customized R_i according to equation (2) gets the best performance of 8.6s, getting 1.73x speedup than the best one (14.87s) among all fixed factors.

6.3 Dual Modes of the CPU Kernel

Adaptive switching between pull and notify-pull modes according to the density of active vertices improves the performance of Garaph-C significantly. We propose an experiment by forcing Garaph-C to run under the two modes for each iteration respectively to illustrate the necessities of the dual-mode abstraction.

Figure 10 shows that the performance gap between notify-pull and pull modes is quite significant. For SSSP, the notify-pull mode outperforms the pull mode in most iterations, except several iterations where most vertices are updated. For CC, the pull mode only outperforms the notify-pull mode at the first few iterations when most of the vertices remain active. However, with switching model proposed in Section 4.2, Garaph-C is able to adopt the better mode for each iteration. We see that the switch of Garaph-C occurs at the next iteration around the intersection of the two modes' performance curves.

	CuSha	Ligra	GraphChi	Gemini	Garaph
enwiki-2013	47.57	70.45	41.1	17.96	26.2
gsh-2015-tpd	-	442	249	107.21	137.6
twitter-2010	-	-	654	266.18	353.8

Table 4: Preprocessing Cost (in seconds) of PR

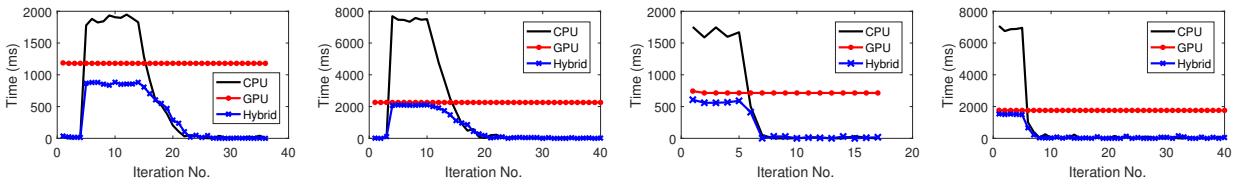
6.4 Scheduling Performance

To demonstrate the speedup of processing graphs on a hybrid platform (compared to processing it on the host only or the GPU only), we run SSSP and CC on twitter-2010 and renren-2010 datasets under CPU-only, GPU-only and hybrid for each iteration, respectively.

As Figure 11 shows, Garaph-H gains much better performance by combing CPU and GPU kernels. For both SSSP and CC, when a few vertices are active, Garaph-H chooses to only use the CPU to process graphs with notify-pull model. However, when most of vertices are active, Garaph-H switches to pull mode in the CPU kernel, and the GPU also joins in computation and accelerates the processing significantly. In contrast, The runtime of the Garaph-C incurs long processing time when most of vertices are active, whereas Garaph-G remains constant because the amount of computation does not change in all iterations of SSSP and CC.

6.5 Preprocessing Cost

Finally, we evaluate the preprocessing cost of Garaph compared to CuSha, Ligra, Gemini and GraphChi on a RAID-0 provided by three SATA SSDs. Garaph and GraphChi will write preprocessed data into secondary storages. Garaph's preprocessing is light-weight, which only needs to scan the input data twice to build the CSC and the CSR data. Table 4 shows the preprocessing cost



(a) SSSP: twitter-2010 (b) SSSP: renren-2010 (c) CC: twitter-2010 (d) CC: renren-2010

Figure 11: Runtime under CPU-only, GPU-only and hybrid modes in two datasets.

of PR on three graphs. It is clear that Garaph's preprocessing is faster than CuSha, Ligra and GraphChi. As Garaph needs to write preprocessed data to secondary storages, Garaph's preprocessing is slower than Gemini.

7 Related Works

In recent years, a large number of graph processing systems have been proposed [36, 28, 37, 7, 16, 11, 32, 29, 21, 17, 10, 2, 27, 22, 35, 33]. We mention here only those most closely related to our work.

GPUs provide a massive amount of parallelism with the potential to outperform CPUs. Numerous graph processing systems [33, 16, 35, 31] have been proposed to use GPUs for high-performance graph processing. Medusa [35] is a generalized GPU-based graph processing framework that focuses on abstractions for easy programming and scaling to multiple GPUs. CuSha [16] primarily focuses on exploring new graph representations to allow faster graph processing. It uses two graph representations G-Shards and Concatenated Windows to improve coalesced memory access and GPU utilization. In CuSha, vertices' values are stored in shards and CuSha needs a phase to write updated values to shards. This design would incur significant data transfer cost between GPU and CPU if it enxtened data to host memory. In Garaph, updated vertices' values are written to global memory instead of shards. Further, CuSha incurs heavy conflicts without any data replicaiton. However, Garaph adopts the replication-based gather to reduce conflicts. Futher, both Medusa and CuSha cannot process graphs exceeding the GPU memory capacity.

To scale out GPU-accelerated graph processing, TOTEM [8] is a processing engine that provides a convenient environment to implement graph algorithms on hybrid CPU and GPU platforms. TOTEM can process graphs whose size is smaller than the host memory capacity. gGraph [34] is another hybrid CPU-GPU system which uses hard-disk drives (HDDs) as secondary storages. For load balancing, both systems initially partition graph into subgraphs that are proportional to the processing power of CPUs and GPUs.

However, existing GPU-accelerated systems cannot fully utilize the GPU for processing large-scale graphs due to ignoring heavy write contention caused by skewed power-law degree distributions and properties of graph algorithms. Garaph further exploits the replication and dynamical scheduling to achieve the best performance on the CPU/GPU hybrid platform.

Shared-memory graph processing systems provide either a push-based [22, 30, 2, 15] or a pull-based model [21, 10, 11, 6, 7], or a switchable model [32, 12, 36]. Garaph modifies push/pull models of Ligra [32] to notify-pull/pull models to achieve lock-free processing. In particular, Ligra's push operations are atomic, whereas our notify-pull/pull model is lock-free with edge-based partitioning. Also, Ligra's critical switching parameter is set by experience, which may not be the best parameter for different applications and datasets. However, notify-pull/pull model is switched by the data-driven model, and thus can achieve a better performance.

8 Conclusion

In this work, we designed a general graph processing platform called Garaph which can efficiently process large-scale graphs using both CPUs and GPUs on a single machine. We design critical system components such as replication-based GPU kernel, optimized CPU kernel with edge-based partition and dual computation modes, and dispatcher with dynamic CPU-GPU scheduling. Our deployment and evaluation reveal demonstrate that Garaph can fully explore both CPU and GPU parallelism for graph processing. Although Garaph is designed for a single machine, the proposed techniques could also be easily applied to distributed, CPU/GPU hybrid systems. Garaph focuses on systems with fast storage (e.g., RAM, or NVM/PCIe-SSD arrary). However, for enviorment with slow secondary storages (e.g., HDD-based system), other optimizations on I/O of secondary storages should be introduced to allivate the bottleneck.

Acknowledgements

Authors would like to thank Christopher J. Rossbach, our shepherd, and the anonymous reviewers for their insightful comments. This work was supported by the National Natural Science Foundation under Grant No. 61472009 and Shenzhen Key Fundamental Research Projects under Grant No. JCYJ20151014093505032.

References

[1] ABOU-RJEILI, A., AND KARYPIS, G. Multilevel algorithms for partitioning power-law graphs. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International* (2006), IEEE, pp. 10–pp.

[2] AVERY, C. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara 11* (2011).

[3] BAKHODA, A., YUAN, G. L., FUNG, W. W., WONG, H., AND AAMODT, T. M. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on* (2009), IEEE, pp. 163–174.

[4] BOLDI, P., CODENOTTI, B., SANTINI, M., AND VIGNA, S. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience 34*, 8 (2004), 711–726.

[5] BOLDI, P., MARINO, A., SANTINI, M., AND VIGNA, S. BUbiNG: Massive crawling for the masses. In *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web* (2014), International World Wide Web Conferences Steering Committee, pp. 227–228.

[6] CHEN, R., DING, X., WANG, P., CHEN, H., ZANG, B., AND GUAN, H. Computation and communication efficient graph processing with distributed immutable view. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing* (2014), ACM, pp. 215–226.

[7] 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.

[8] GHARAIBEH, A., REZA, T., SANTOS-NETO, E., COSTA, L. B., SALLINEN, S., AND RIPEANU, M. Efficient large-scale graph processing on hybrid cpu and gpu systems. *arXiv preprint arXiv:1312.3018* (2013).

[9] GOMEZ-LUNA, J., GONZALEZ-LINARES, J. M., BENITEZ, J. I. B., AND MATA, N. G. Performance modeling of atomic additions on gpu scratchpad memory. *IEEE Transactions on Parallel and Distributed Systems 24*, 11 (2013), 2273–2282.

[10] 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.

[11] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. Graphx: Graph processing in a distributed dataflow framework. In *OSDI* (2014), vol. 14, pp. 599–613.

[12] HAN, W., MIAO, Y., LI, K., WU, M., YANG, F., ZHOU, L., PRABHAKARAN, V., CHEN, W., AND CHEN, E. Chronos: a graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 1.

[13] HARRIS, M., ET AL. Optimizing parallel reduction in cuda. *NVIDIA Developer Technology 2*, 4 (2007).

[14] INTEL. Intel ssd dc p3608 series product brief. Tech. rep., 2015.

[15] 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 *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), ACM, pp. 169–182.

[16] 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* (2014), ACM, pp. 239–252.

[17] KYROLA, A., BLELLOCH, G. E., GUESTRIN, C., ET AL. Graphchi: Large-scale graph computation on just a pc. In *OSDI* (2012), vol. 12, pp. 31–46.

[18] LEHMBERG, O., MEUSEL, R., AND BIZER, C. Graph structure in the web: Aggregated by pay-level domain. In *Proceedings of the 2014 ACM conference on Web science* (2014), ACM, pp. 119–128.

[19] LESKOVEC, J., AND KREVL, A. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[20] LESKOVEC, J., LANG, K. J., DASGUPTA, A., AND MAHONEY, M. W. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics 6*, 1 (2009), 29–123.

[21] 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. *Proceedings of the VLDB Endowment 5*, 8 (2012), 716–727.

[22] 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.

[23] MISLOVE, A., MARCON, M., GUMMADI, K. P., DRUSCHEL, P., AND BHATTACHARJEE, B. Measurement and Analysis of Online Social Networks. In *Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC'07)* (San Diego, CA, October 2007).

[24] NVIDIA. Kepler gk110 architecture whitepaper, v1.0. Tech. rep., 2012.

[25] NVIDIA. *CUDA C Programming Guide*. nvidia, 2017.

[26] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The pagerank citation ranking: Bringing order to the web. Tech. rep., Stanford InfoLab, 1999.

[27] PINGALI, K., NGUYEN, D., KULKARNI, M., BURTSCHER, M., HASSAAN, M. A., KALEEM, R., LEE, T.-H., LENHARTH, A., MANEVICH, R., MÉNDEZ-LOJO, M., ET AL. The tao of parallelism in algorithms. In *ACM Sigplan Notices* (2011), vol. 46, ACM, pp. 12–25.

[28] 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), ACM, pp. 410–424.

[29] 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.

[30] SALIHOGLU, S., AND WIDOM, J. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management* (2013), ACM, p. 22.

[31] SEO, H., KIM, J., AND KIM, M.-S. Gstream: A graph streaming processing method for large-scale graphs on gpus. In *ACM SIGPLAN Notices* (2015), vol. 50, ACM, pp. 253–254.

[32] 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.

[33] 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 *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2016), ACM, p. 11.

[34] ZHANG, T., ZHANG, J., SHU, W., WU, M.-Y., AND LIANG, X. Efficient graph computation on hybrid cpu and gpu systems. *The Journal of Supercomputing 71*, 4 (2015), 1563–1586.

[35] ZHONG, J., AND HE, B. Medusa: Simplified graph processing on gpus. *IEEE Transactions on Parallel and Distributed Systems 25*, 6 (2014), 1543–1552.

[36] ZHU, X., CHEN, W., ZHENG, W., AND MA, X. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)(Savannah, GA* (2016).

[37] ZHU, X., HAN, W., AND CHEN, W. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX Annual Technical Conference* (2015), pp. 375–386.

GPU Taint Tracking

Ari B. Hayes
Rutgers University

Lingda Li*
Brookhaven National Lab

Mohammad Hedayati
University of Rochester

Jiahuan He
Rutgers University

Eddy Z. Zhang
Rutgers University

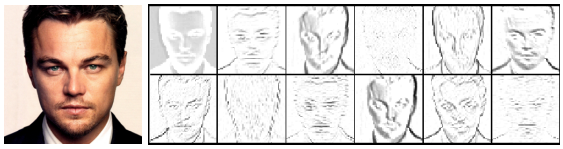
Kai Shen
Google

Abstract

Dynamic tainting tracks the influence of certain inputs (taint sources) through execution and it is a powerful tool for information flow analysis and security. Taint tracking has primarily targeted CPU program executions. Motivated by recent recognition of information leaking in GPU memory and GPU-resident malware, this paper presents the first design and prototype implementation of a taint tracking system on GPUs. Our design combines a static binary instrumentation with dynamic tainting at runtime. We present new performance optimizations by exploiting unique GPU characteristics—a large portion of instructions on GPU runtime parameters and constant memory can be safely eliminated from taint tracking; large GPU register file allows fast maintenance of a hot portion of the taint map. Experiments show that these techniques improved the GPU taint tracking performance by 5 to 20 times for a range of image processing, data encryption, and deep learning applications. We further demonstrate that GPU taint tracking can enable zeroing sensitive data to minimize information leaking as well as identifying and countering GPU-resident malware.

1 Introduction

GPUs have been widely used in many important application domains beyond scientific computing, including machine learning, graph processing, data encryption, computer vision, etc. Sensitive information propagates into GPUs and, while being processed, leaves traces in GPU memory. For example, in a face recognition application, besides the input photo itself, the features extracted at different levels of the deep learning neural networks may also contain part of sensitive or private information. Figure 1 shows the extracted features from the first level of neural networks in a face recognition program, where Figure 1(a) is the original picture and Figure 1(b) are features such as silhouette of a human face. Given a sensitive input user photo, features in deep learning applications may contain much of the sensitive data as well. Other sensitive data in today's GPU applications include encryption keys, digits in personal checks, license plates, location information in virtual reality apps, etc. If not tracked or protected, sensitive information can be inadvertently leaked or stolen by malicious applications on GPUs.



(a) Org. Photo (b) Extracted Features

Figure 1: Neural network information leaking example.

Taint analysis [3, 4, 6, 11, 22, 26, 29, 30] is a powerful tool for information flow tracking and security. It tracks where and how sensitive information flows during program execution. Taint analysis is a form of data flow analysis, wherein an input set of sensitive data is marked as "tainted", and this taint is tracked during runtime as it spreads into different locations in memory via move, arithmetic, and control operations. Taint analysis results can be used to protect data by clearing tainted variables at the end of its life range—for instance, the temporary key schedule at every round of AES algorithm—or by encrypting live but inactive tainted data [27]. Taint analysis can also help identify and counter abnormal behaviors of malicious malware. Existing dynamic taint analysis has primarily been applied to CPU programs though its functions are increasingly desirable for GPUs as well.

This paper presents the first design and implementation of a GPU taint tracking system. Our approach is based on static binary instrumentation that enables dynamic taint tracking of a GPU program. In comparison

*This work was done when Lingda Li was a postdoctoral associate at Rutgers University

to dynamic instrumentation that captures and modifies instructions on the fly, our approach does not require a dynamic instrumentation framework or virtual machine emulation that is not readily available on GPUs. We perform static instrumentation on GPU program binaries without source access so that it is easy to apply in practice. We instrument programs on a per-application basis and when the program runs, every thread can dynamically track information flow by itself.

The major challenge for efficient taint tracking is that tracking every dynamic binary instruction is expensive. Our solution exploits the fact that a large portion of a typical GPU program execution operates on un-taintable runtime parameters and constants. Examples include the logical thread indexes, thread block identifiers, dimension configurations, and pointer-type kernel parameters. We use a simple filtering policy that the runtime taint tracking only operates on instructions whose operands can be reached from potential global memory taint sources through dependencies and can reach potential global memory taint sinks. We present an iterative two-pass taint reachability analysis to implement such instruction filtering which significantly reduces runtime taint tracking costs.

Our taint tracking system also exploits the heterogeneous memory architecture on GPUs. A GPU has different types of memory, including either physically partitioned or logically partitioned memory storage. For instance, local memory is private to every thread, shared memory is a software cache visible to a group of threads, and global memory is visible to all threads. Our taint system handles different types of memory storage separately and optimizes the tracking for different types of memory storage. Specifically, we allocate a portion of the register file to store part of the taint map, since GPU contains a much larger register file than CPU—e.g., every streaming multi-processor (SM) has 64K registers on most NVIDIA GPUs. Not all registers are needed [8, 20] nor the maximum occupancy is necessary [10] for best performance. Using fast access registers to maintain the taint map of frequently accessed data will improve the dynamic tainting performance.

GPU taint tracking enables data protection that clears sensitive (tainted) data objects at the end of their life range as well as detects leak of the sensitive data in the midst of program execution. We recognize that data in different GPU memory storage may have different life ranges. For instance, registers and local memory are thread private and can be cleared once a thread finishes its execution; shared memory is only used by a thread block and sensitive data in shared memory can be cleared by that thread block once it releases the SM. Global memory may be accessed at any time of a program run so we cannot clear it at the end of every kernel execution. However, we can detect when and where the sensitive information (in global memory) is sent out by instrumenting memory communication APIs since all communication between GPU, CPU and other network devices require explicit memory API calls. By checking if the sensitive information falls within the region of memory that is transferred, we can identify GPU malware (like Keylogger [17] and Jellyfish [15]) that uses GPU to snoop CPU activities while storing these activities in GPU memory. Such GPU-resident malware would escape detection by a CPU-only taint tracking mechanism.

2 Background

GPU functions, also known as kernel functions, make use of memory which is not directly accessible from the CPU. GPU memory is split into several regions, both on-chip and off-chip. On-chip memory consists of registers, caches, and scratch-pad memory (called *shared memory* in NVIDIA terminology). Note that we use NVIDIA terminology throughout this paper. Off-chip memory is GDDR SGRAM, which is logically distributed into texture memory, constant memory, local memory, and global memory, with texture memory and constant memory mapped to texture and constant caches.

Both texture memory and constant memory are read-only during the GPU kernel execution. Therefore, in this paper we focus on registers, shared memory, local memory, and global memory. Shared memory is available to the programmer, often treated as a software cache. Local memory is thread-private, and is most commonly used for register spilling. Global memory is visible to the entire GPU device, and is typically used as input and output for GPU functions. In all four of these memory types, data persists after deallocation [25].

Global memory can be set and cleared through API functions, with overhead similar to that of running a GPU kernel, but local memory, shared memory, and registers are only accessible from within a kernel function, and allocated and deallocated by the driver. These three memory types can only be reliably cleared through instrumentation. Moreover, local memory and registers are managed by compilers and they can only be cleared by compile-time instrumentation.

Sensitive information can also propagate to different data storage locations on GPU: memory, software caches, and registers. An example is the advanced encryption standard (AES), in which the key and the plain text to be encrypted may reside in different types of memory [25]. They can be stored in global memory as allocated data objects and in registers as program execution operands.

Currently, there is less memory protection on GPUs as compared with CPUs. When two applications run si-

multaneously on the same GPU with the Multi-process Service (MPS), one application can peek into the memory of another application, documented in NVIDIA's MPS manual at Section 2.3.3.1, "*An out-of-range read in a CUDA Kernel can access CUDA-accessible memory modified by another process, and will not trigger an error, leading to undefined behavior*." When two applications do not run simultaneously, in which case every application will get a serially scheduled time-slice on the whole GPU, information leaking is still possible. The second running application can read data left by the first running application if its allocated memory locations happen to overlap with those of the first one. This vulnerability has been detailed in several recent works [18, 25, 28].

Future hardware trends such as the fine-grained memory protection in AMD APUs suggest potentially better process isolation. Hardware-level memory protection may exhibit superior performance, but its realization must take into account the hardware implementation complexity. And more importantly, process memory protection does not distinguish sensitive data and its propagation within one program or process. Such protection would be critical for securing sensitive information flows between CPU, GPU and their memories.

3 Efficient GPU Taint Tracking

A typical information flow tracking system on CPUs monitors instructions and operands to maintain proper taint propagations. For example, in a binary operation $v = binop\ v_1, v_2$, assuming $T(v_1)$, $T(v_2)$, and $T(v)$ represent the taint status for operands v, v_1, and v_2 respectively: true means tainted and false means untainted. The taint tracking rule for this instruction is $T(v) = T(v_1) \mid\mid T(v_2)$. Taint statuses for all data storage locations (program memory, registers, conditional flags, etc.) are maintained in a taint map in memory. A baseline GPU taint tracking system would operate in a similar way.

Dynamic taint tracking [3, 4, 6, 11, 22, 26, 29, 30] is known to incur high runtime costs. Fortunately, GPU executions exhibit some unique characteristics that enable optimization. We present an optimization that recognizes and identifies the large portion of GPU instructions that cannot be involved in taint propagation from sources to sinks. Furthermore, given the large register file on GPU and frequent register accesses, we maintain register taint map in registers to accelerate their taint tracking. These optimizations are performed through binary-level static analysis.

3.1 Taint Reachability

On GPUs, we discover that programs frequently operate on a set of critical runtime un-taintable values, and that not all operands need to be tracked. We exploit this fact and only track the operands that potentially carry taints or may have an impact on the state transition of the un-taintable objects. In the earlier example, if v_1 does not carry any taint, the taint maintenance only needs to track v_2 and v such that $T(v) = T(v_2)$. If neither v_1 or v_2 can be tainted, or if v does not propagate to memory, no taint maintenance is necessary for the variables v, v_1, and v_2.

A frequently used GPU runtime un-taintable is the logical thread index. A thread index is used to help identify the task that is assigned to every thread. It is a built-in variable, and does not come from global memory that is managed by a programmer, and thus the instruction operand as a thread index or an expression of thread index can never be tainted. Similarly, other built-in thread identification variables, including thread block id and dimension configuration, are also un-taintable.

Another frequently used GPU runtime un-taintable value are the non-scalar pointer-type kernel parameters. A GPU kernel function does not allow call-by-reference. To reference a memory data object that can be modified at runtime, it can only use pointers. Moreover, these kernel parameters are kept in a memory region named as "constant memory" in GPUs and are read-only in kernel execution. The memory region pointed to by the kernel parameter must be tracked, but the pointer or the address expression computed using the pointer and thread index (or part of the expression) does not need to be tracked. Other examples include compile-time un-taintable values, such as loop induction variables and stack framework pointers, programmer-specified constants, and combinations of GPU-specific runtime constants with these constants. We analyze and categorize these un-taintable values in Section 5 and Table 1.

To avoid tracking un-taintable values in GPU programs, we take the following approaches.

1. We classify an instruction operand into two types: taintable and un-taintable. The *taintable* state indicates that the operand might be tainted at runtime—whether it will be really tainted depends on the exact dynamic analysis done by tracking instructions. The *un-taintable* state indicates that the operand cannot be tainted at runtime. Any operand that cannot be reached from the taintable source is un-taintable. The taintable sources are program inputs given by the users and reside in the global memory on GPUs. Examples include face recognition photos, a plain-text message, and encryption key.

2. A variable can be overwritten with taintable or un-taintable values at different program execution points. We check for potential state transition of a variable: from un-taintable to taintable, or from taintable to un-taintable. The latter arises in a situa-

tion called *taint removal*—e.g., assigning a constant to a register who might be in a tainted state before the assignment but must now transition to the un-tainted state.

3. We statically check the memory reachability: whether an operand might reach memory (potential taint sinks). Even if an operand is taintable, as long as it does not flow into memory, it will not affect any taint sink. We do not need to add tracking instruction for this type of operands. Common examples include loop trip counters, predicate registers, and stack frame pointers.

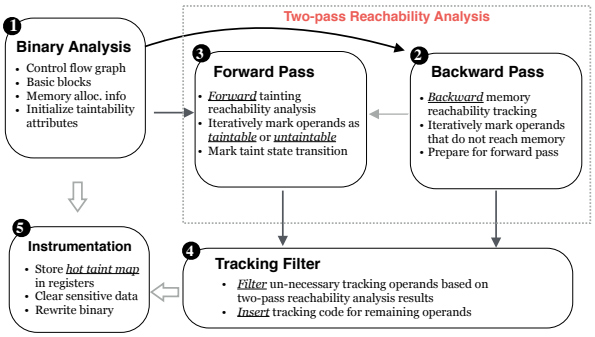


Figure 2: Overview of our taint tracking system.

Original Code	Tracking Code
1:block0:	1:
2: R0 = 0x1234;	2:
3: R0 = 0x0;	3: T(R0) = false;
4: if (some_condition)	4:
5: R0 = [R1];	5: T(R0) = T([R1]) || T(R1);
6: [R2] = R0;	6: T([R2]) = T(R0);
7: some_condition = random();	7:
8: GOTO block0;	8:

We show an example in the code snippet above. The code describes a loop. Register R0 is overwritten with different types of values. Initially R0 is written with an un-taintable value (lines 2-3). Later in the *if* statement, it is written by a taintable value [R1]; note that here the [R1] notation indicates a memory operation and the address of the memory location is R1. We need a tracking instruction within the *if* statement since [R1] comes from global memory and every operand from memory needs to be tracked. We do not need a tracking instruction for line 2 since $0x1234$ is a constant and the assignment target R0 at line 2 cannot reach memory. However, we do need a tracking instruction for line 3 since the assignment target may reach memory and taint removal applies here (R0 may be tainted from an earlier iteration of the loop and if so, taint must be removed here).

3.2 Iterative Two-pass Taint Analysis

To mark the taintability and reachability attributes for every operand and to detect potential taint state transition, we perform an iterative dataflow analysis.

There are two passes in our iterative dataflow analysis component. The forward pass marks the taintable operands and the un-taintable operands only at the program points where a potential taint state transition occurs. The backward pass marks an operand that potentially reaches memory (taint sinks). In the end, when adding code to track the original program, we only track the operands that are marked in both forward and backward passes.

Figure 2 provides an overview of our taint tracking system. First, we analyze the binary code to obtain the control flow graph and a list of basic blocks. A basic block is the maximum length single-entrance and single-exit code segment. We also mark the operands that are known to be un-taintable before the program starts. They include built-in thread identification variables, non-scalar pointer type kernel parameters, and other programmer-specified constants.

We perform the backward pass first to analyze each operand and set its memory reachability attribute. We name it the *mightSpread* attribute, indicating whether there exists an execution path through which the value of this operand might spread into memory.

We then perform the forward pass to mark all operands as taintable or un-taintable, and for every un-taintable operand, we also analyze if its last immediate state is taintable in one of the potential execution paths. If an operand is taintable or its last immediate state is taintable, we set the *taintTrack* attribute to be true. The taintTrack attribute indicates that the operand may be tainted at runtime. For an indirect memory operand, we also need an attribute on the taintability of the addressing register. We call this *addrTrack* attribute.

Finally, in the *Tracking Filter* component, we scan all instructions and review the taintability and reachability attributes each operand. For the destination operand, if its taintTrack and mightSpread attributes are both true, we add tracking code for this destination operand, otherwise we don't. Similarly, for source operands, if both of its taintTrack and mightSpread attributes are true, we add tracking code for the source operand before the tracking code for the destination operand. For an indirect memory source operand, if its addrTrack and mightSpread attributes are both true, we add taint tracking code for the source operand addressing register.

We describe the detailed algorithms for forward and backward passes below.

Forward Taint Reachability Analysis The input is a control flow graph and a set of basic blocks for the GPU program. The output is the taintTrack property value for every operand in every instruction. We show the forward

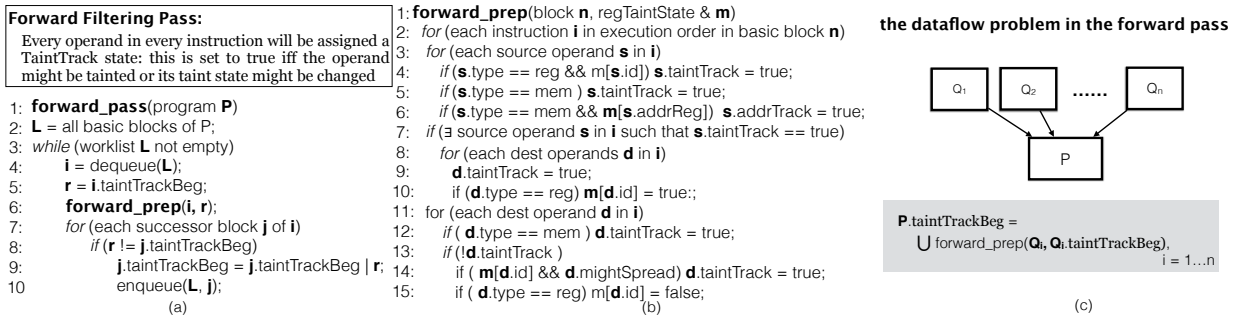


Figure 3: Forward taint reachability analysis.

pass algorithm in Figure 3(a).

We adopt the fixed-point computation algorithm that is used in standard dataflow analysis (DFA) framework. Function *forward_pass* in Figure 3*(a)* scans the basic blocks one by one, sets the taintTrack attribute for every operand, and updates the *taintTrackBeg* attribute for every basic block. Our forward analysis pass checks if one basic block's taintability updates affect another basic block's taintability results, and if so, adds the affected basic block to the worklist. Initially, all basic blocks are added to the list. The analysis pass finishes only when all basic block's taintability results do not change.

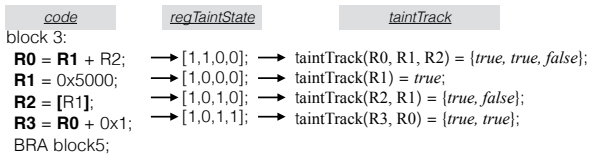
A DFA problem is formulated using (a set of) dataflow equation(s). We describe the dataflow equation as follows. The taintTrackBeg attribute describes the taint tracking state of every register at the beginning of a basic block, which is a bit array. Every bit in the bit array corresponds to one physical register. If a register's taintTrack attribute is true at the beginning of the basic block of interest, this bit is set to 1, otherwise 0. Assume a basic block P and it has n predecessor basic blocks Q_i, $i = 1...n$, the dataflow relation is

$$\text{P.taintTrackBeg} = \cup \text{forward_prep}(Q_i, Q_i.\text{taintTrackBeg}).$$

The *forward_prep* function in Figure 3(b) updates the taintability state for all instructions in a basic block based on the taintability state at the beginning of the basic block. It scans the first instruction to the last instruction.

Given an instruction, the *forward_prep* function checks its source operands first (lines 3−6 in Figure 3(b)). If a source operand is register and the taintTrack attribute is true, this source operand needs to be tracked. If a source operand is of memory type, it has to be tracked. Note that if the address register of an indirect memory operand is taintable, we need to track the register as well—setting addrTrack attribute at line 6 in Figure 3(b).

Next, the *forward_prep* function checks every destination operand. If any source operand needs to be tracked based on the above analysis, destination operand needs to be tracked as well. In the meantime, we update the register tracking state for the corresponding destination operand (line 10 in Figure 3(b)). If the destination operand is of memory type, it needs to be tracked. If the destination operand is un-taintable (lines 13-15 in Figure 3), and its prior tracking state is taintable, and the destination operand might spread to memory, the destination operand needs to be tracked as well. Further we update the register tracking state for the corresponding destination operand.



We use the above example to illustrate the **forward_prep** step for updating the register tracking state. Let the initial regTaintState be [0, 1, 0, 0], meaning that only register R1 is found to be taintable on entry to this basic block. Since the first instruction has R1 as a source and R0 as a destination, we set the operand's taintTrack flag and regTaintState[0] to true.

Since the second instruction writes an immediate value to R1, but since regTaintState[1] was previously true, we have to set the operand's taintTrack flag to true, if its result can spread to memory. This instruction potentially changes the taint value of R1 at runtime from true to false, so if it can reach memory, then we need to instrument it, or else we will suffer from over-tainting as a result of incorrectly treating the data as still being tainted. We flip regTaintState[1] to false since at compile-time and at the second instruction, register R1 is untaintable.

The next instruction loads from memory into R2, so we set the operands' taintTrack flags and regTaintState[2] to true, because memory is a possible taint source. The final instruction before the branch carries potential taint from R0 to R3; since regTaintState[0] is true, regTaintState[3] is set to true along with the operand's taintTrack flag.

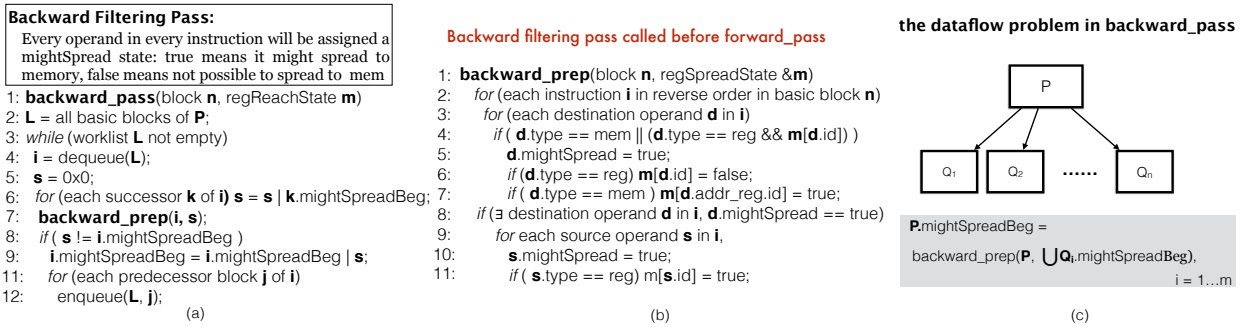


Figure 4: Backward memory reachability analysis.

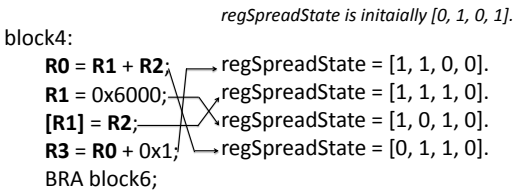
Backward Memory Reachability Analysis Similar to the forward pass, the backward pass uses the program as input. The output is the memory reachability property of every operand. The backward reachability analysis also uses a dataflow analysis framework, which solves the *mightSpreadBeg* bit array for every individual basic block, representing the memory reachability state of the registers at the beginning of basic block. In this bit array, each bit corresponds to one physical register. A value of 1 for the bit at index n of basic block b means that the value of register Rn at the beginning of basic block b might reach memory.

The relationship between one basic block P and its successor basic blocks $Q_i, i = 1..m$, where m is the total number of immediate successor basic blocks, is described using the following equation:

$$P.\text{mightSpreadBeg} = \text{backward_prep}(\cup Q_i.\text{mightSpreadBeg}).$$

The initial mightSpreadBeg bit array is set to 0 for every basic block. Our backward pass keeps updating the mightSpreadBeg bit arrays until they do not change any further (Figure 4(a)). In the meantime, the attribute mightSpread is updated for every operand, as described in Figure 4(b).

The *backward_prep* function calculates mightSpreadBeg for every individual basic block. In Figure 4(b), we scan the instructions in reverse order in a basic block. First, we check the destination operand, if it is register type and the register's memory reachability state is true, the destination operand's mightSpread attribute is set to true. In the meantime, we update the register's memory reachability state for the destination register to false since the value to spread into memory is defined at this point and for any instruction that happens before this instruction, they don't see the same value as defined here. If it is memory type, the mightSpread attribute is set to true and the address register's reachability state is set to true (line 7 in Figure 4(b)). Next, we check the source operands. If any destination operand can spread into memory, then all source operands' mightSpread property is set to true (line 10). Correspondingly, we will set the register reachability state to true (line 11).



We use the above example to illustrate the process for updating the mightSpreadBeg bit arrays in the backward pass. The backward pass is mechanically similar to the forward pass, aside from the direction in which instructions are processed. In this example, we assume that registers R1 and R3 have been determined to spread to memory in later blocks, hence the initial regSpreadState value of [0, 1, 0, 1]. We skip over the branch instruction since it has no operands except for a jump offset.

The last instruction has data flow into R3 from R0, and regSpreadState[3] is true, so we mark the R0 operand's mightSpread flag as true and set regSpreadState[0] to true. We also flip regSpreadState[3] to false since this instruction is overwriting R3.

The instruction before the last stores register R2 to memory, so we simply mark the R2 operand's mightSpread flag as true and set regSpreadState[2] to true.

The third instruction counting from the last puts an immediate value into R1, so we set regSpreadState[1] to false. Finally, the fourth instruction counting from the last has data flow into R0 from R1 and R2, and regSpreadState[0] is true, so we mark both source operands' mightSpread as true, set regSpreadState[1] and regSpreadState[2] to true, and set regSpreadState[0] to false since R0 has been overwritten.

3.3 Register Taint Map in Registers

A GPU contains a much larger register file than CPU does—e.g., every streaming multi-processor has 64K registers on most NVIDIA GPUs. Registers are naturally accessed frequently and maintaining their taint statuses require frequent reads and writes from/to their taint map locations. At the same time, the large GPU register

file presents the opportunity to maintain a portion of the taint map in registers. These facts motivate us to place the register taint map in registers.

We use multiple 32-bit general purpose registers to store the taint map, in which one bit corresponds to one register that is tracked. Using register-stored taint map increases the number of registers used per-thread, and might decrease occupancy, determined as the number of active threads running at the same time. Fortunately in many GPU programs, not all the register file is needed [8, 20] nor the maximum occupancy is necessary [10] for the best program performance. Therefore the overall taint tracking cost is significantly reduced by our use of register-stored taint map, as demonstrated later in evaluation.

4 Tainting-Enabled Data Protection

Taint tracking results can be used to help protect sensitive data and prevent information leaking on GPUs. We describe two major use cases of taint tracking analysis and present our prototype implementation of tainting-enabled data protection.

4.1 Sensitive Data Removal

Lee et al. [18] and Pietro et al. [25] have recently demonstrated that information leaking from one program to another may occur in GPU local memory between GPU kernel executions, and in GPU global memory between program runs. Our taint tracking results may help a program understand the propagation of certain sensitive information and clear all taints before relevant points of vulnerability (e.g., clearing local memory taints at the end of each kernel and clearing global memory taints at the end of program run).

We make a prototype implementation of this use case. For registers, we let every thread clear its own tainted registers. It is possible that some threads exit earlier than others. However since register taint map is thread-private, we can insert the clearing code right before every *EXIT* point and thus early-exiting threads can also clear their tainted registers early. For local memory, since it is thread-private, we treat it the same way as registers. Note that registers and local memory cannot be cleared by programmers themselves (unlike shared memory and global memory) and thus a trustworthy binary instrumentation tool is necessary to prevent sensitive data from leaving taints on GPUs.

For shared memory, since shared memory is visible to all threads in the same basic block, we need to make sure the sensitive shared memory data is cleared *after* all threads in the same thread block finish their work. Therefore, our design is to create a control flow reconvergence point for all threads since different threads might take different execution paths. We then insert a thread block level barrier at the reconvergence point before clearing the tainted shared memory data.

Pietro et al. [25] proposed a register-spilling based attack, which makes use of compiler to force spilling the registers so that the encryption key (or reversibly transformed encryption key) in the AES encryption module in the *SSLShader* program can be moved from registers to local memory. Then a second running application can steal the leaked information in local memory. Our taint clearing approach prevents such attacks by clearing the registers, local memory, and shared memory right before every thread in the GPU application completes.

Experimental results in Section 5 will show that the data clearing cost is low—worst-case slowdown of 13% and in most cases no more than 5% slowdown.

4.2 GPU Malware Countermeasure

GPU taint analysis identifies where and when sensitive data is sent from GPU device to CPU or other network devices. This is especially important for integrated CPU-GPU whole system taint tracking. A dynamic taint tracking system that only monitors data dependences during CPU execution may miss the influence propagation of untrusted inputs or execution results through GPU computation. For example, GPU malware Keylogger [17] and Jellyfish [15] exploited direct memory access (DMA) at mapped CPU memory to snoop the CPU system activities and steal host information. GPU may obtain the leaked CPU information, process it, and send it through a network or other output device while evading countermeasures that only monitor CPU executions.

Our GPU data protection system can not only clear sensitive data, but also capture possible attempts of stealing and emitting sensitive information. We prevent this type of attacks by dynamically monitoring the data transfer between CPU and GPU. If the GPU-mapped CPU memory contains sensitive information (i.e., keystroke buffer in the Keylogger attack [17]), the mapped data region is marked as taint sources. We track the dependency propagation of tainted data in GPU executions. Further we statically instrument memory transfer APIs so that before any data is sent from GPU through cudaMemcpy APIs in CUDA or clEnqueueReadBuffer APIs in OpenCL, the memory address range is checked. If the transmitted data falls within the sensitive tainted memory range, we either alert the system that tainted data is transmitted, or mark the corresponding CPU destination memory region (if data is transferred back to CPU) as tainted. Since all communication between GPU and other devices rely on explicit memory transfer API, we can check and protect information flow by instrumenting these memory transfer APIs.

Our taint tracking and data protection system helps

protect applications that utilize both CPU and GPU, if combined with CPU taint tracking. Our work ensures full system taint tracking that is essential to whole system security. We are not aware of any other work that provides the same degree of protection. Besides the Keylogger case, other potential whole-system tracking examples include a web site that relies on taint tracking to prevent untrusted user inputs with malicious database queries (e.g., through SQL injections) or invoking dangerous system calls (e.g., through buffer overflow attacks).

Finally, when GPU tainting is securely applied to untrusted programs, it can also identify malicious programs that attempt to scan uninitialized data which may have been left by previous kernel and program runs from other users. We have not implemented this type of data protection. However, our tool can be readily extended to help detect uninitialized memory region containing sensitive data left by prior GPU program execution and clear / zero them if appropriate.

5 Evaluation

We perform evaluation on a machine configured with an NVIDIA GTX 745. This is a "Maxwell" generation GPU with compute capability 5.0. Since NVIDIA's compiler and binary ISA are closed-source, we modify the GPU binaries using tools inspired by the asfermi [12] and MaxAs [9] projects, allowing for binary instructions be directly inserted into the executable.

5.1 Benchmarks

Our evaluation employs a variety of GPU kernels in deep learning, image processing, and data encryption. First, Caffe [16] is a deep learning framework in which a user writes a "prototxt" file describing the input and layers of the deep learning network (e.g., convolutional layers, inner-product layers, etc.), which can be fed into the Caffe executable to create, train, and test the network. Newer versions of Caffe allow various layers to be executed on the GPU via CUDA. A common use of Caffe is image classification. We use three Caffe kernels in our evaluation: im2col, ReLUForward, and MaxPoolForward. These three kernels consume the majority of the execution time for image classification.

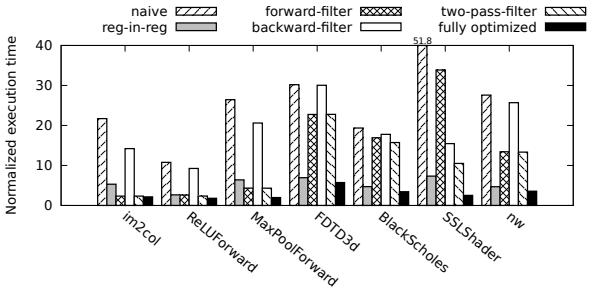
We additionally use kernel functions from the CUDA SDK [23], the Rodinia benchmark suite [2], and SSLShader [13]. From the CUDA SDK we include BlackScholes, a program for financial analysis, and FDTD3d, a 3D Finite Difference Time Domain solver. As a numerical analysis program, FDTD3d is unlikely to have sensitive data to protect, but serves as an additional data point for testing our performance. From Rodinia, we include Needleman-Wunsch, a bioinformatics benchmark used for DNA sequencing. From SSLShader, we include an AES encryption program.

5.2 Taint Analysis & Optimizations

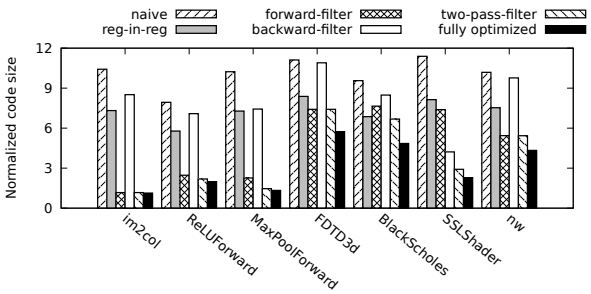
We evaluate the effectiveness of the two performance enhancing techniques in Section 3—taint reachability filtering and taint map in registers.

Since we modify the executable directly, we measure the cost of taint analysis in terms of both slowdown and code size. There are a few factors which exacerbate these costs. Whenever we insert an instruction to get or set a location's taintedness in memory, we first have to calculate its address. Since addresses for global memory are 64-bit on this architecture, but registers and integer operations are 32-bit, this requires multiple instructions with immediate dependencies.

Additionally, each thread has access to only one carry flag, so if it is already in use where we need to get the taint address, extra instructions are needed to spill it into a register or to memory. Furthermore, the singular carry flag makes it difficult to interleave instructions effectively, since they may overwrite each other's result. As the GPU is incapable of out-of-order execution within a thread, the latency for accessing the taint-map is costly.



(a) Normalized slowdown from instrumentation.



(b) Normalized code size after instrumentation.

Figure 5: Overhead of tainting instrumentation.

Figure 5(a) illustrates the GPU tainting slowdown with each of our optimizations, compared to native execution. The 'naive' bar shows slowdown without any optimizations, the 'reg-in-reg' bar shows the results of placing part of the taint map into registers, the 'forward-filter' and 'backward-filter' bars show the results of each filter pass, the 'two-pass-filter' bar shows results when using

both filter passes, and the 'fully optimized' bar shows results when using all of these optimizations. Figure 5(b) shows normalized code sizes (static instruction counts) for the same cases.

Figure 5 shows that both two-pass filtering and hot register taint map can reduce the tainting cost significantly. For the filter passes, there is a high correlation between relative slowdown and code size after instrumentation. Saving taint mapping into registers does not shrink as much of the code size as two-pass filtering, but it still improves the tracking performance significantly. The tracking cost saving comes more from the reduced memory latency than from reduced instruction count.

Taint Map in Registers Even on its own, saving part of the taint map in registers reduces significant time during taint analysis. The main alternatives, local memory and global memory, are both off-chip memories that may take hundreds of cycles to access. Even the cache to which such memory is saved is off-chip, because the on-chip L1-cache is typically only used for read-only data on newer architectures [24]. Since most GPU programs have numerous threads running at once, some of this latency is hidden by some threads continuing to execute while others wait for memory accesses to complete, but even so, saving register taint information into registers reduces slowdown compared to naive taint tracking in our benchmarks by 78% on average.

Filtering The forward pass filtering also saves significant time, though it has more variance across different benchmarks. Its effectiveness stems from the properties of GPU kernels. Most kernels make use of non-taintable, read-only data such as thread ID and grid size to perform many calculations. Additionally, function parameters are read-only in GPU functions, making it impossible for them to become tainted in most programs. On its own, the forward pass reduces slowdown in our benchmarks by an average of 53%.

kernel	parameter	immediate	const mem.	thread block id
im2col	85%	85%	29%	64%
ReLUForward	20%	40%	47%	57%
MaxPoolForward	70%	72%	57%	58%
FDTD3d	17%	17%	11%	12%

Table 1: Percentage of filtered-out instructions for various reasons.

We also analyze the reason why we are able to filter out a significant number of instructions for some applications in the forward pass. Table 1 shows the percentage of filtered out instructions under different categories. *Parameter* means one or more source registers are from the (constant-memory) kernel parameters. *Immediate* means one or more source operands are immediate numbers. *Const memory* means at least one source is from constant memory. Finally, *thread/block id* means the influence is from the identifier of the current thread or thread block. The identifiers are stored in special registers private to each thread or constant memory depending on the architecture, but in either case they are known at static-time. While it might be surprising that the sum of percentages due to multiple reasons may exceed 100%, note that an instruction may be filtered out due to multiple reasons.

We discover that most instructions are filtered out because of these four categories. The reason is that GPU programs distribute workload among threads based on their ids. To get the assigned workload, each thread must perform a lot of computation using ids, immediate, and constant memory values (e.g., thread block & grid dimensions). The computation results, together with parameters (e.g., the start address of an array), are used to fetch assigned data. Then the real computation starts as well as the taint tracking. For most GPU programs, the real computation is short with several instructions, and the preprocessing including address calculation consumes most of the time. That is why we can filter out most instructions in our forward pass: most instructions do preparation work and are not related to the potentially tainted input data. For FDTD3d, the computation is more complex and fewer instructions are filtered out. It also explains why FDTD3d does not benefit from two-pass filtering as much as compared with other benchmarks, as shown in Figure 5(a).

The backward pass is usually less effective than the other optimizations. While a lot of the inputs to a kernel function are effectively constants, the only means of returning anything is through global memory. As such, we can expect that most operations will produce values which influence memory. Regardless, the backward pass does provide some benefit in most cases, and in the SSLShader benchmark it reduces slowdown compared to the naive approach by 22%.

Combined Optimizations Compared to the forward pass, the two-pass filter reduces slowdown by 12% on average, and compared to the backward pass, it reduces slowdown by 50%. Full optimization reduces slowdown by 56% compared to the two-pass filter, and 42% compared to only keeping part of the taint map in registers. This demonstrates the merit of combining our different optimization techniques, which together reduce slowdown by an average of 87%.

With full optimizations, our benchmarks' kernel functions experience an average normalized runtime of 3.0× after instrumentation. The FDTD3d benchmark suffers the worst slowdown at 5.7× runtime, due to frequent use of shared memory making the filter less effective. The

Needleman-Wunsch benchmark, which also has shared memory usage, is the next slowest with a 3.6× runtime. Although the SSLShader benchmark also makes use of shared memory, it only uses shared memory to store compile-time constants for faster retrieval, allowing us to filter out all shared memory instructions for less runtime slowdown of 2.5×.

One special consideration when modifying GPU programs is occupancy—the number of threads that can be live at once. A high occupancy means that latency is less costly, as the GPU can switch to different groups of threads every cycle. Since our instrumentation results in additional use of registers, and the register file is evenly split among all live threads, there is potential for occupancy to be decreased, hurting performance more drastically. In such a case, it may be more beneficial not to store any part of the taint map into registers. However, in practice, we use few enough additional registers that reducing occupancy is unlikely, since for every 32 registers in the original program, we only need 1 extra register to store their taintedness. We find that GPU programs typically use less than 64 registers per-thread, and so none of our benchmarks require more than two extra registers per-thread for storing register taintedness.

5.3 Memory Protection

We next evaluate the incorporation of memory protection into our dynamic analysis framework. As discussed in Section 4, the GPU does not clear memory before deallocation. This includes all types of memory, both on-chip and off-chip. [25] demonstrates that data left behind even in local memory and shared memory can be stolen, such as the encryption key and plaintext in the SSLShader benchmark. We have found that this data can also be stolen directly from registers by preparing a kernel function with the same thread block size and occupancy as the victim kernel function—thereby ensuring the register file will be partitioned in the same, predictable manner—and then manually coding the eavesdropping kernel's binary to read the desired registers.

Programmers can manually erase global memory before program exit, but registers and local memory are allocated by the compiler and cannot be as easily cleared. Sensitive data in registers, local memory, and also shared memory must be cleared before the kernel function exits, or else a malicious kernel function may be invoked and acquire these resources for itself. We leverage our instrumentation framework to clear sensitive data in these regions, via additional modification to the binary code. This can be used to prevent attacks such as the one in [25], which stole encryption key data through such resources. The results are summarized in Table 2.

Since registers and local memory are thread-private, they can be safely cleared by each thread prior to exit.

GPU kernel	*Memory*	*Slowdown*
im2col	N/A	0.26%
ReLUForward	N/A	0.33%
MaxPoolForward	N/A	0.59%
FDTD3d	Shared	5.10%
BlackScholes	N/A	0.40%
SSLShader	Local	0.41%
needle	Shared	13.05%

Table 2: Slowdown from memory erasure during kernel execution, measured as a fraction of the original kernel time. "Memory" column indicates which memory types need to be cleared (besides registers).

We insert instructions to clear this data before the EXIT instruction, using the results of our forward-filter pass to avoid unnecessary work. But shared memory is shared by every thread in a thread block, and therefore may not be safe to erase until all of its threads finish execution. Before the EXIT instruction we insert a synchronization barrier, which causes threads to halt until all other threads in the block reach the same point, and then add a loop which has every thread zero out a separate portion of shared memory. In benchmarks with less regular control flow, where threads exit at different points in the code, we can instead have shared memory cleared by a subset of its threads.

We find that the cost to clear tainted registers is trivial, adding only a fraction of a percent to runtime. Each register takes only one cycle of amortized time to erase for every 32 threads, and the GPU is likely able to overlap most of these cycles with memory stalls from other threads. None of our benchmarks use local memory by default, since it is usually used for register spilling. In order to evaluate the slowdown of clearing local memory, we recompile SSLShader, which uses 40 registers, to instead use 20 registers. Clearing local memory and registers in this benchmark adds 0.41% time overhead.

Shared memory is slower to clear. In FDTD3d, clearing taints in shared memory adds 5.10% runtime compared to the original kernel function, and in Needleman-Wunsch it adds 13.05%. The increased slowdown compared to clearing local memory likely stems from the use of a loop, due to the GPU's inability to perform speculative and out-of-order execution, forcing a thread to wait until each shared memory location is cleared until it can zero the next one. Local memory is simpler to handle, with every thread accessing the same logical addresses despite using different physical locations, allowing for the local memory clearing loop to be fully unrolled.

Using the taint information to erase only sensitive data can help significantly, compared to naively clearing these memories fully. For example, in the SSLShader bench-

mark the tainted registers and local memory are cleared in 47 mSecs, but this benchmark makes use of shared memory which is never tainted. If its shared memory arrays are erased, in addition to clearing the small amount of registers and local memory in their entirety, then the overhead would jump to 407 mSecs.

6 Related Work

Dynamic taint analysis [3, 4, 6, 11, 22, 26, 29, 30] tracks data (and sometimes control) dependencies of information as a program or system runs. Its purpose is to identify the influence of taint sources on data storage locations (memory, registers, etc.) during execution. Taint tracking is useful for understanding data flows in complex systems, detecting security attacks, protecting sensitive data, and analyzing software bugs. Its implementation usually involves static code transformation, dynamic instrumentation, or instruction emulation using virtual machines to extend the program to maintain tainting metadata. While existing dynamic tainting systems track CPU execution, this paper presents the first design and implementation of a GPU taint tracking system.

A large body of previous work presented techniques to improve the performance of CPU taint tracking. LIFT [26] checks whether unsafe data are involved before a code region is executed, and if not, no taint tracking code is executed for that code region to reduce overhead. Minemu [1] proposes a novel memory layout to reduce the number of taint tracking instructions. It also uses SSE registers for taint tracking to reduce performance overhead. TaintEraser [30] makes use of function summary to reduce the performance overhead of taint tracking. It summarizes taint propagation at the function level so that instruction level taint tracking is reduced. TaintDroid [6] is a taint analysis tool proposed for Android systems. By leveraging Androids virtualized execution environment and coarse-grained taint propagation tracking, it can achieve nearly real time analysis with low performance overhead. Jee et al. [14] proposed to separate taint analysis code from the original program, and dynamic and static analysis was applied on the taint analysis code to optimize its performance. In this paper, we present new performance optimizations by exploiting unique GPU characteristics.

Security vulnerabilities on GPUs have been recognized recently. Dunn et al. [5] showed that sensitive data can be leaked into graphics device driver buffers. They proposed encryption to protect data in transit over the device driver but their approach does not protect data in GPU memory. Lee et al. [18] uncovered several vulnerabilities of leaking sensitive data in GPU memory—leaking global memory data after a program context finishes and releases memory without clearing; leaking local memory data across kernel switches on a CU. They did not present any solution to address these vulnerabilities. More recently, Pietro et al. [25] proposed memory zeroing to prevent information leaking in GPU. However, memory zeroing alone provides limited protection—it cannot track information flow in memory; nor can it counter GPU malware such as Keylogger [17] and Jellyfish [15]. Furthermore, GPU tainting is complementary to memory zeroing—tainting identifies a subset of sensitive memory for zeroing to reduce the costs.

GPU information flow analysis has been performed in the past. Leung et al. [19] and Li et al. [21] employed static taint analysis to reduce the overhead of GPU program analysis and verification. Static analysis requires memory aliasing analysis of memory accesses that are inherently imprecise. While they are suitable for testing and debugging purposes [19,21], security data flow analysis in this paper requires more precise dynamic tracking. Farooqui et al. [7] proposed static dependency analysis between thread index and control conditions to identify possible thread divergence in GPU executions (the result of which helps determine whether symbolic execution can be performed on given GPU basic blocks). Their static dependency analysis is narrowly targeted and it is unclear whether it applies to general taint tracking.

7 Conclusion

Recent discoveries of information leaking through GPU memory and GPU-resident malware call for systematic data protection in GPUs. This paper presents the first design and implementation of a dynamic taint tracking system for GPU programs. We exploit unique characteristics of GPU programs and architecture to optimize taint tracking performance. Specifically, we recognize that a large portion of instructions on GPU runtime parameters and constants can be safely eliminated from taint tracking to reduce tainting costs. We also utilize the large GPU register file for fast maintenance of the taint map for registers. These optimizations result in 5 to 20 times tainting speed improvement for a range of image processing, data encryption, and deep learning applications.

Acknowledgement

We thank Adam Bates for his help during the preparation of the final version of the paper, and the anonymous reviewers for their insightful comments. This work is supported by NSF Grant NSF-CCF-1421505, NSF-CCF-1628401, and the Google Faculty Award. 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] BOSMAN, E., SLOWINSKA, A., AND BOS, H. Minemu: The world's fastest taint tracker. In *International Workshop on Recent Advances in Intrusion Detection* (2011), Springer, pp. 1–20.

[2] 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 *IEEE Int'l Symp. on Workload Characterization (IISWC)* (2009), pp. 44–54.

[3] CHOW, J., PFAFF, B., GARFINKEL, T., CHRISTOPHER, K., AND ROSENBLUM, M. Understanding data lifetime via whole system simulation. In *Proc. of the 13th USENIX Security Symp.* (2004), pp. 321–336.

[4] CLAUSE, J., LI, W., AND ORSO, A. Dytan: A generic dynamic taint analysis framework. In *Proc. of the 2007 Int'l Symp. on Software Testing and Analysis* (London, United Kingdom, 2007), pp. 196–206.

[5] DUNN, A. M., LEE, M. Z., JANA, S., KIM, S., SILBERSTEIN, M., XU, Y., SHMATIKOV, V., AND WITCHEL, E. Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels. In *Proc. of the 10th USENIX Symp. on Operating Systems Design and Implementation (OSDI)* (Hollywood, CA, Oct. 2012), pp. 61–75.

[6] ENCK, W., GILBERT, P., HAN, S., TENDULKAR, V., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. Taintdroid: An information-flow tracking system for real-time privacy monitoring on smartphones. *ACM Trans. Comput. Syst. 32*, 2 (June 2014), 5:1–5:29.

[7] FAROOQUI, N., SCHWAN, K., AND YALAMANCHILI, S. Efficient instrumentation of GPGPU applications using information flow analysis and symbolic execution. In *Proc. of Workshop on General Purpose Processing Using GPUs* (Salt Lake City, UT, Mar. 2014), GPGPU-7, pp. 19:19–19:27.

[8] GEBHART, M., KECKLER, S. W., KHAILANY, B., KRASHINSKY, R., AND DALLY, W. J. Unifying primary cache, scratch, and register file memories in a throughput processor. In *Proc. of the 45th Annual IEEE/ACM Int'l Symp. on Microarchitecture* (Vancouver, B.C., CANADA, Dec. 2012), MICRO-45, pp. 96–106.

[9] GRAY, S. Maxas: Assembler for nvidia maxwell architecture. `github.com/NervanaSystems/maxas`, 2014.

[10] HAYES, A. B., AND ZHANG, E. Z. Unified on-chip memory allocation for simt architecture. In *Proc. of the 28th ACM Int'l Conf. on Supercomputing* (Munich, Germany, 2014), ICS'14, pp. 293–302.

[11] HO, A., FETTERMAN, M., CLARK, C., WARFIELD, A., AND HAND, S. Practical taint-based protection using demand emulation. In *Proc. of the First EuroSys Conf.* (Leuven, Belgium, Apr. 2006), pp. 29–41.

[12] HOU, Y., LAI, J., AND MIKUSHIN, D. Asfermi: An assembler for the nvidia fermi instruction set. `code.google.com/p/asfermi/`, 2011.

[13] JANG, K., HAN, S., HAN, S., MOON, S. B., AND PARK, K. SSLShader: Cheap SSL acceleration with commodity processors. In *Proc. of the 8th USENIX Conf. on Networked Systems Design and Implementation (NSDI)* (Boston, MA, Mar. 2011), pp. 1–14.

[14] JEE, K., PORTOKALIDIS, G., KEMERLIS, V. P., GHOSH, S., AUGUST, D. I., AND KEROMYTIS, A. D. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In *Proc. of the 19th Annual Network & Distributed System Security Symp. (NDSS)* (San Diego, CA, Feb. 2012).

[15] GPU rootkit PoC by team Jellyfish. `github.com/x0r1/jellyfish`.

[16] 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).

[17] LADAKIS, E., KOROMILAS, L., VASILIADIS, G., POLYCHRONAKIS, M., AND IOANNIDIS, S. You can type, but you cant hide: A stealthy GPU-based keylogger. In *Proc. of the 6th European Workshop on Systems Security (EuroSec)* (Prague, Czech Republic, Apr. 2013).

[18] LEE, S., KIM, Y., KIM, J., AND KIM, J. Stealing webpages rendered on your browser by exploiting GPU vulnerabilities. In *Proc. of the 35th IEEE Symp. on Security and Privacy* (San Jose, CA, May 2014), pp. 19–33.

[19] LEUNG, A., GUPTA, M., AGARWAL, Y., GUPTA, R., JHALA, R., AND LERNER, S. Verifying GPU kernels by test amplification. In *Proc. of the 33rd ACM Conf. on Programming Language Design and Implementation (PLDI)* (Beijing, China, June 2012), pp. 383–394.

[20] LI, C., YANG, Y., LIN, Z., AND ZHOU, H. Automatic data placement into GPU on-chip memory resources. In *proc. of the 13th Int'l Symp. on Code Generation and Optimization (CGO)* (Feb. 2015), pp. 23–33.

[21] LI, P., LI, G., AND GOPALAKRISHNAN, G. Practical symbolic race checking of GPU programs. In *Proc. of SC14: The Int'l Conf. for High Performance Computing, Networking, Storage and Analysis* (New Orleans, LA, Nov. 2014).

[22] NEWSOME, J., AND SONG, D. X. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of the 12th Annual Network & Distributed System Security Symp. (NDSS)* (San Diego, CA, 2005).

[23] NVIDIA. GPU computing sdk. `developer.nvidia.com/gpu-computing-sdk`.

[24] NVIDIA. Maxwell tuning guide. `docs.nvidia.com/cuda/maxwell-tuning-guide/`, 2014.

[25] PIETRO, R. D., LOMBARDI, F., AND VILLANI, A. CUDA leaks: A detailed hack for CUDA and a (partial) fix. *ACM Trans. on Embedded Computing Systems (TECS) 15*, 1 (Feb. 2016).

[26] QIN, F., WANG, C., LI, Z., SEOP KIM, H., ZHOU, Y., AND WU, Y. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *39th Int'l Symp. on Microarchitecture* (2006), pp. 135–148.

[27] TANG, Y., AMES, P., BHAMIDIPATI, S., BIJLANI, A., GEAMBASU, R., AND SARDA, N. CleanOS: Limiting mobile data exposure with idle eviction. In *Proc. of the 10th USENIX Symp. on Operating Systems Design and Implementation (OSDI)* (Hollywood, CA, Oct. 2012), pp. 77–91.

[28] VASILIADIS, G., ATHANASOPOULOS, E., POLYCHRONAKIS, M., AND IOANNIDIS, S. PixelVault: Using GPUs for securing cryptographic operations. In *Proc. of the 21st ACM Conf. on Computer and Communications Security (CCS)* (Scottsdale, Arizona, USA, Nov. 2014), pp. 1131–1142.

[29] XU, W., BHATKAR, S., AND SEKAR, R. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proc. of the 15th USENIX Security Symp.* (Vancouver, B.C., Canada, 2006).

[30] ZHU, D. Y., JUNG, J., SONG, D., KOHNO, T., AND WETHERALL, D. TaintEraser: Protecting sensitive data leaks using application-level taint tracking. *SIGOPS Oper. Syst. Rev. 45*, 1 (Feb. 2011), 142–154.

Optimizing the Design and Implementation of the Linux ARM Hypervisor

Christoffer Dall Shih-Wei Li Jason Nieh
Department of Computer Science
Columbia University
{*cdall,shihwei,nieh*}*@cs.columbia.edu*

Abstract

Modern hypervisor designs for both ARM and x86 virtualization rely on running an operating system kernel, the hypervisor OS kernel, to support hypervisor functionality. While x86 hypervisors effectively leverage architectural support to run the kernel, existing ARM hypervisors map poorly to the virtualization features of the ARM architecture, resulting in worse performance. We identify the key reason for this problem is the need to multiplex kernel mode state between the hypervisor and virtual machines, which each run their own kernel. To address this problem, we take a fundamentally different approach to hypervisor design that runs the hypervisor together with its OS kernel in a separate CPU mode from kernel mode. Using this approach, we redesign KVM/ARM to leverage a separate ARM CPU mode for running both the hypervisor and its OS kernel. We show what changes are required in Linux to implement this on current ARM hardware as well as how newer ARM architectural support can be used to support this approach without any changes to Linux other than to KVM/ARM itself. We show that our redesign and optimizations can result in an order of magnitude performance improvement for KVM/ARM, and can provide faster performance than x86 on key hypervisor operations. As a result, many aspects of our design have been successfully merged into mainline Linux.

1 Introduction

Given their customizability and power efficiency, ARM CPUs have become an attractive option across a wide range of computer systems, from their dominance in mobile and embedded systems to their increasing popularity in server systems. Recognizing that virtualization is a key technology for the successful deployment of ARM hardware, modern ARM CPUs include hardware support for virtualization, the Virtualization Extensions (VE). Popular ARM hypervisors, including KVM [14] and Xen [30], utilize VE to run unmodified commodity operating systems (OSes) and applications across a wide range of deployment scenarios for virtualization, from enterprise servers to locomotive computer systems [5]. Despite these successes, we have shown that ARM virtualization costs remain too high for important deployment scenarios, including network-intensive workloads such as network functions virtualization (NFV) [14, 12].

Hypervisor designs for ARM and x86 virtualization rely on running a full OS kernel to support the hypervisor functionality. This is true for both Type 1 hypervisors which run an isolated hypervisor runtime and Type 2 hypervisors which integrate with a host OS [17]. KVM, a Type 2 hypervisor, is integrated with the Linux kernel and leverages the Linux kernel for common OS functionality such as scheduling, memory management, and hardware support. Similarly, Xen, a Type 1 hypervisor, runs a full copy of Linux in a special privileged Virtual Machine (VM) called Dom0 to leverage existing Linux drivers to provide I/O for other VMs. These *hypervisor OS kernels* which support the hypervisor run in the CPU's kernel mode just like OS kernels run when not using virtualization. Modern hypervisors use hardware support for virtualization, avoiding the need to deprivilege the guest OS kernel in a VM to run in user mode [8]. As each VM runs a guest OS kernel in addition to the hypervisor OS kernel, and both kernels run in the same kernel mode, the shared hardware state belonging to kernel mode is multiplexed among the OS kernels. When a VM is running on the CPU, the VM's guest OS kernel is using the CPU's kernel mode, but when it becomes necessary to run the hypervisor, for example to perform I/O on behalf of the VM, the hypervisor OS kernel takes over using the CPU's kernel mode.

Transitioning from the guest OS kernel to the hypervisor OS kernel involves saving the guest kernel's state and restoring the hypervisor kernel's state, and vice versa. This save and restore operation is necessary because both the guest and hypervisor OS kernels use the same hardware state such as registers and configuration settings, but in different contexts. On x86, these transitions happen using operations architecturally defined as part of the Intel Virtual Machine Extensions (VMX). These hardware operations save and

restore the entire kernel mode register state, typically as a result of executing a single instruction. Unlike x86, ARM does not provide a hardware mechanism to save and restore kernel mode state, but instead relies on software performing these operations on each register, which results in much higher overhead. The cost of transitioning from a VM to the hypervisor can be many times worse on ARM than x86 [12].

To address this problem, we present a new hypervisor design and implementation that takes advantage of unique features of the ARM architectural support for virtualization in the context of Type 2 hypervisors. We take a fundamentally different approach that runs the hypervisor together with its OS kernel in a separate CPU mode from kernel mode. ARM VE provides an extra hypervisor CPU mode, EL2, designed to run standalone hypervisors. EL2 is a separate mode from the EL1 kernel mode, and the architecture allows switching from EL1 to EL2 without saving or restoring any EL1 register state. In this design, the hypervisor and its OS kernel no longer run in EL1, but EL1 is reserved exclusively to be used by VMs. This means that the kernel mode hardware state no longer has to be multiplexed between the hypervisor OS kernel and a VM's guest OS kernel, and transitioning between the two does not require saving and restoring any kernel mode state. This new design, using separate hardware state for VMs and the hypervisor OS kernel can significantly improve hypervisor performance.

Our new hypervisor design benefits from the Virtualization Host Extensions (VHE) introduced in ARMv8.1. With VHE, our design does not require any changes to existing hypervisor OS kernels. Without VHE, our design requires modifications to the hypervisor OS kernel so it can run in EL2 instead of EL1. Although Type 1 hypervisors also suffer from poor performance due to slow transitions between the hypervisor OS kernel and guest OS kernels, our design is not easily applicable to Type 1 hypervisors. We focus on improving the performance of Type 2 hypervisors on ARM given their widespread popularity, which is at least in part due to their benefits over Type 1 hypervisors on ARM. ARM hardware does not have the same legacy and standards as x86, so Type 1 hypervisors have to be manually ported to every hardware platform they support. Type 2 hypervisors leverage their host OS and are automatically supported on all hardware platforms supported by their host OS.

Running the hypervisor and its OS kernel in a separate CPU mode with its own hardware state allows a number of improvements to the hypervisor implementation. First, transitioning from the VM to the hypervisor no longer requires saving and restoring the kernel mode register state. Second, the hypervisor OS kernel can program hardware state used by the VM directly when needed, avoiding extra copying to and from intermediate data strutures. Third, the hypervisor and its OS kernel no longer need to operate across different CPU modes with separate address spaces which requires separate data structures and duplicated code. Instead, the hypervisor can directly leverage existing OS kernel functionality while at the same time configure ARM hardware virtualization features, leading to reduced code complexity and improved performance.

We have implemented our approach by redesigning KVM/ARM and demonstrated that it is effective at providing significant performance benefits with reduced implementation complexity. A number of our changes have been merged into mainline Linux over the course of Linux kernel versions v4.5 through v4.8, with additional changes scheduled to be applied in upcoming kernel versions. We show that our redesign and optimizations can result in an order of magnitude performance improvement for KVM/ARM in microbenchmarks, and can reduce virtualization overhead by more than 50% for real application workloads. We show that both hardware and software need to work together to provide the optimal performance. We also show that our optimized KVM/ARM provides significant performance gains compared to x86, indicating that our hypervisor design combined with the required architectural support for virtualization provides a superior approach to x86 hardware virtualization.

2 Background

We first provide a brief overview of current state-of-the-art Type 2 hypervisor designs on both x86 and ARM and discuss how they must multiplex kernel mode to run both their VM and hypervisor OS kernels using hardware virtualization support. For architectural support for virtualization on x86, we focus on Intel VMX, though AMD-V is similar for the purposes of this discussion.

2.1 Intel VMX

The Intel Virtual Machine Extensions (VMX) [21], support running VMs through the addition of a new feature, *VMX operations*. When VMX is enabled, the CPU can be in one of two VMX operations, VMX root or VMX non-root operation. Root operation allows full control of the hardware and is for running the hypervisor. Non-root operation is restricted to operate only on virtual hardware and is for running VMs. VMX provides memory virtualization through Extended Page Tables (EPT) which limits the memory the VM can access in VMX non-root. Both VMX root and non-root operation have the same full set of CPU modes available to them, including both user and kernel mode, but certain sensitive instructions executed in non-root operation cause a transition to root operation to allow the hypervisor to maintain complete control of the system. The hypervisor OS kernel runs in root operation and a VM's guest OS kernel runs in non-root operation, but both run in the same CPU mode. Since the hypervisor and the VM have separate execution contexts in form of register state and configuration state, all of this state must be multiplexed between root and non-root operation.

VMX supports this multiplexing in hardware by defining two VMX transitions, VM Entry and VM Exit. VM Entry transitions from root to non-root operation which happens when the hypervisor decides to run a VM by executing a specific instruction. VM Exit transitions from non-root to root operation which transfers control back to the hypervisor on certain events such as hardware interrupts or when the VM attempts to perform I/O or access sensitive state. The transitions are managed by hardware using an in-memory data structure called the Virtual-Machine Control Structure (VMCS). VMX root and non-root operation do not have separate CPU hardware modes, but VMX instead multiplexes the modes between the hypervisor and VM by saving and restoring CPU state to memory using hardware VMX transitions.

2.2 ARM VE

ARM took a very different approach than x86 in adding hardware virtualization support. Instead of introducing an orthogonal feature to distinguish between the hypervisor and VM operation, ARM extended the existing CPU mode hierarchy, originally just EL0 user mode and EL1 kernel mode, by adding a separate more privileged mode called EL2 to run the hypervisor. Although ARM refers to EL0, EL1, and EL2 as exception levels, we refer to them here as CPU modes to simplify the discussion. EL2 cannot be used to run existing unmodified OS kernels for a number of reasons. For example, EL2 has its own set of control registers and has a limited and separate address space compared to EL1, so it is not compatible with EL1. Furthermore, EL2 does not easily support running userspace applications in EL0 which expect to interact with a kernel running in EL1 instead of EL2.

Therefore, both the hypervisor and VM OS kernels must run in EL1, and this mode must be multiplexed between the two execution contexts. On ARM, this can be done by software running in EL2. EL2 is a strictly more privileged mode than user and kernel modes, EL0 and EL1, respectively, and EL2 has its own execution context defined by register and control state, and can therefore completely switch the execution context of both EL0 and EL1 in software, similar to how the kernel in EL1 context switches between multiple userspace processes running in EL0.

When both the hypervisor and VM OS kernels run at the same privilege level on ARM without an equivalent feature to x86 VMX operations, an obvious question is how to differentiate between the roles of the hypervisor and the VM kernel. The hypervisor kernel should be in full control of the underlying physical hardware, while the VM kernel should be limited to the control of virtual hardware resources. This can be accomplished by using ARM VE which allows fine grained control of the capabilities of EL1. Software running in EL2 can enable certain sensitive instructions and events executed in EL0 or EL1 to trap to EL2. For example, similar to x86 EPT, ARM VE provides memory virtualization by adding an additional stage of address translation, the stage 2 translations. Stage 2 translations are controlled from EL2 and only affect software executing in EL1 and EL0. Hypervisor software running in EL2 can therefore completely disable the stage 2 translations when running the hypervisor OS kernel, giving it full access to all physical memory on the system, and conversely enable stage 2 translations when running VM kernels to limit VMs to manage memory allocated to them.

ARM VE supports the multiplexing of EL1 analogously to how EL0 is multiplexed between processes using EL1. Because EL2 is a separate and strictly more privileged mode than EL1, hypervisor software in EL2 can multiplex the entire EL1 state by saving and restoring each register and configuration state, one by one, to and from memory. In line with the RISC design of ARM, and in contrast to the CISC design of x86, ARM does not provide any hardware mechanism to multiplex EL1 between the hypervisor and VM kernels, but instead relies on existing simpler mechanisms in the architecture. For example, if a VM kernel tries to halt the physical processor, because this is a sensitive instruction and the VM is not allowed to control the physical CPU resource, this instruction will cause a trap to the more privileged EL2 mode, which can then reuse existing instructions to save and restore state and switch the EL1 execution context to the hypervisor kernel context, configure EL1 to have full access to the hardware, and return to EL1 to run the hypervisor OS kernel.

2.3 KVM

Figure 1 compares how the KVM hypervisor runs using x86 VMX versus ARM VE. We refer to the hypervisor OS kernel as the host OS kernel, the more commonly used term with KVM, and applications interacting directly with the OS, and running outside of a VM, as host user space. Figure 1(a) shows how KVM x86 works. The hypervisor and host OS run in root operation, with the host user space running in the least privileged CPU mode level 3, and the host kernel running in the privileged CPU mode, level 0, similar to running on a native system. All of the VM runs in non-root operation and the VM user space and kernel also run in level 3 and level 0, respectively. Transitions between root and non-root mode are done in hardware using the atomic VMX transitions, VM Entry and VM Exit.

Figure 1(b) shows how KVM/ARM works. Since the host OS kernel cannot run in EL2, but EL2 is needed to enable the virtualization features and to multiplex EL1, KVM/ARM uses split-mode virtualization [14] to support both the host OS kernel running in EL1 and at the same time run software in EL2 to manage the virtualization features and multiplex EL1. Most of the hypervisor functionality runs in EL1 with full access to the hardware as part of the host OS kernel, and a small layer, the *lowvisor*, runs in EL2.

When KVM x86 runs a VM, it issues a single instruction to perform the VM Entry. The VM Entry operation saves the

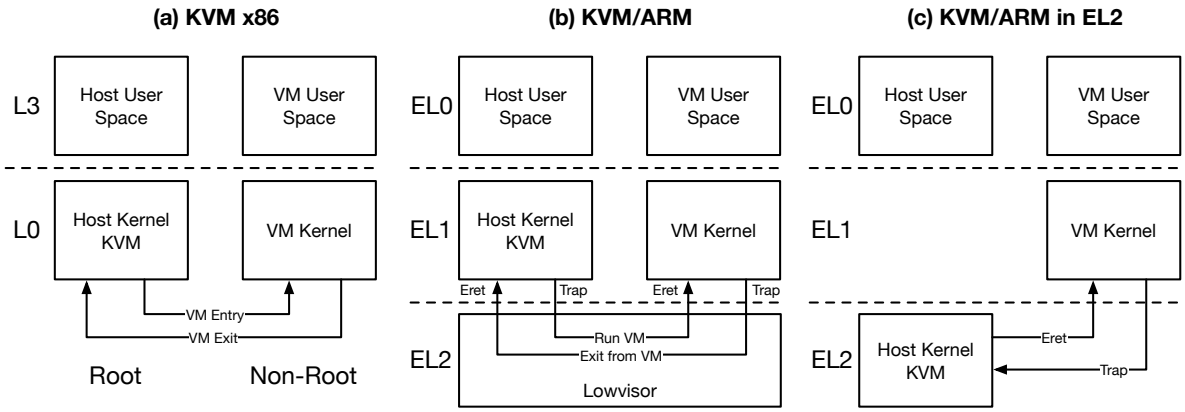


Figure 1: Hypervisor Designs and CPU Privilege Levels

hypervisor execution context of the processor to the VMCS and restores the VM execution context from the VMCS. On a VM Exit, x86 VMX performs the reverse operation and returns to the hypervisor. Since ARM does not have a single hardware mechanism to save and restore the entire state of the CPU, KVM/ARM issues a hypercall to trap to the lowvisor in EL2, which saves and restores all the registers and configuration state of the CPU, one by one, using a software defined structure in memory. After changing the EL0 and EL1 execution context to the VM, the lowvisor performs an *exception return* (eret) to the VM. When the VM traps to EL2, the lowvisor again saves and restores the entire state of the CPU and switches the execution context of EL0 and EL1 back to the hypervisor. As we shall see in Section 5.1, while the x86 VMX transitions are very complicated hardware operations, and the traps on ARM from EL1 to EL2 are cheap, multiplexing the kernel mode between two contexts ends up being much more expensive on ARM as a result of having to save and restore the entire CPU state in software.

3 Hypervisor OS Kernel Support

Running the hypervisor OS kernel in the same CPU mode as the VM kernels invariably results in multiplexing the kernel CPU mode, either in hardware or software, which adds overhead from the need to save and restore state. If instead a dedicated separate CPU mode were available to run the hypervisor OS kernel, this would avoid the need to multiplex a single mode and allow the hardware to simply trap from the VM to the hypervisor OS kernel to manage the underlying hardware and service the VM. Being able to transition back and forth between the full hypervisor functionality and the VM quickly without repeatedly saving and restoring the entire CPU state can reduce latency and improve virtualization performance.

Running the hypervisor OS kernel in a separate mode requires support from both hardware and software. The hardware must obviously be designed with a separate mode in addition to the mode used to run the VM kernel and VM user space. The hardware for the separate mode must support running full OS kernels that interact with user space applications. Furthermore, the hypervisor software must be designed to take advantage of running the hypervisor OS kernel in a separate CPU mode. As explained in Section 2, x86 does not meet these requirements because it does not have a separate CPU mode for the hypervisor OS kernel. ARM at least provides a separate CPU mode, EL2, but it was not designed for running hypervisor OS kernels. We show how this limitation can be overcome.

Figure 1(c) shows how KVM/ARM can be re-designed to run both the hypervisor (KVM) and its hypervisor OS kernel (Linux) together in EL2. This design is superior to previous ARM hypervisor designs including existing KVM/ARM and Xen on ARM, because it allows for very fast transitions between the VM and the hypervisor, including when running the hypervisor OS kernel, because there is no need to repeatedly save and restore the entire CPU state when transitioning between the VM and the hypervisor OS kernel. Furthermore, because the hypervisor is integrated with its hypervisor OS kernel, it can directly manage the underlying hardware using existing functionality such as device drivers in the hypervisor OS kernel without having to run special privileged VMs as is the case on Xen [12].

However, running an existing OS kernel in EL2 requires modifying the hardware or OS kernel, because EL2 was designed only to run hypervisors and lacks key features available in EL1, ARM's kernel mode, used to support OS kernels. First, EL2 uses a separate set of control registers accessed using different instructions than the EL1 control registers, causing incompabilities with a kernel written to run in EL1. Second, EL2 lacks support for host user space, which is needed to run applications such as QEMU, which provides device emulation. Running host user space applications in EL0 in conjunction with software running in EL2 without using EL1, as shown in Figure 1(c), requires handling exceptions from EL0 directly to EL2, for example to handle system calls, hardware interrupts, and page faults.

EL2 provides a *Trap General Exceptions* (TGE) bit to configure the CPU to route all exceptions from EL0 directly to EL2, but setting this bit also disables the use of virtual memory in EL0, which is problematic for real applications. Finally, EL2 uses a different page table format and only supports a single virtual address range, causing problems for a kernel written to use EL1's page table format and EL1's support for two separate virtual address space ranges.

3.1 Virtualization Host Extensions

To run existing hypervisor OS kernels in EL2 with almost no modifications, ARM introduced the Virtualization Host Extensions (VHE) in ARMv8.1. VHE is an architectural hardware modification that provides improved support for Type 2 hypervisors on ARM. It provides three key features.

First, VHE introduces additional EL2 registers to provide the same functionality available in EL1 to software running in EL2. VHE adds new virtual memory configuration registers, a new context ID register used for debugging, and a number of new registers to support a new timer. With these new registers in place, there is a corresponding EL2 system register for each EL1 system register. VHE then transparently changes the operation of instructions that normally access EL1 system registers to access EL2 registers instead when they run in EL2. By transparently changing the operation of the instructions, existing unmodified OSes written to issue EL1 system register instructions will instead access EL2 system registers when run in EL2. VHE also changes the bit layout of some EL2 system registers to share the same layout and semantics as their EL1 counterparts.

Second, VHE supports running host user space applications that use virtual memory in EL0 and interact directly with a kernel running in EL2. VHE introduces new functionality so that the EL0 virtual memory configuration can be managed by either EL1 or EL2, depending on a run time configuration setting, which allows EL2 to route exceptions from EL0 directly to EL2 and at the same time support virtual memory in EL0. VHE extends the functionality of the TGE bit such that when enabled and exceptions from EL0 are routed to EL2, virtual memory support is enabled in EL0 and controlled using EL2 page table registers. A Type 2 hypervisor will typically configure EL0 to use the EL2 system registers when running the hypervisor, and configure EL0 to use the EL1 system registers when running the VM.

Third, VHE changes the page table format of EL2 to use the same format as used in EL1, which avoids the need to change an existing OS kernel's page table management code to support different formats. VHE also adds support to EL2 for an additional separate virtual address space which can be used to provide the same split between kernel and user space addresses commonly used by existing ARM OSes in EL1 and EL0.

Using VHE to run Linux as the hypervisor OS kernel in conjunction with KVM requires very little effort. The early boot code in Linux simply sets a single bit in a register to enable VHE, and the kernel itself runs without further modification in EL2.

While the hypervisor OS kernel can run largely unmodified in EL2, the hypervisor itself must be modified to run with VHE. In particular, because EL1 system register access instructions are changed to access EL2 registers instead, the hypervisor needs an alternative mechanism to access the real EL1 registers, for example to prepare a VM's execution context. For this purpose, VHE adds new instructions, the `_EL12` instructions, which access EL1 registers when running in EL2 with VHE enabled. The hypervisor must be modified to replace all EL1 access instructions that should continue to access EL1 registers with the new `_EL12` access instructions when using VHE, and use the original EL1 access instructions when running without VHE.

3.2 *el2*Linux

Unfortunately, VHE hardware is not yet publicly available and remains an optional extension to the ARM architecture. As an alternative, we introduce *el2*Linux [11], a lightly modified version of Linux that runs in EL2 on non-VHE hardware. *el2*Linux brings the benefits of running Linux as the hypervisor OS kernel in a separate CPU mode to existing hardware alongside the KVM hypervisor. It involves three main kernel modifications to Linux.

First, to control its own CPU mode, Linux must access EL2 register state when running in EL2, and we modify the Linux kernel source code as needed to access EL2 system registers instead of EL1 registers. This can be done using either build time conditionals or at runtime using instruction patching to avoid overhead from introducing additional conditional code paths in the kernel.

Second, to support host user space applications such as QEMU in EL0 interacting with a kernel running in EL2, we install a tiny runtime in EL1, which includes an exception vector to forward exceptions to EL2 by issuing a hypercall instruction. The result is that exceptions from EL0 are forwarded to EL2 via EL1. However, this introduces two sources of additional overhead for applications running outside of a VM. One is a small overhead from going through EL1 to EL2 when handling an exception in EL0. The other is a larger overhead due to the need to multiplex EL1 between the EL1 runtime and a VM's guest OS kernel. While saving and restoring the EL1 state is expensive, it is only necessary when running host user space applications, not on each transition between a VM and the hypervisor. For the KVM hypervisor, returning to host user space is already an expensive transition on both ARM and x86. As a result, KVM is heavily optimized to avoid returning to host user space. Measurements presented in Section 5 indicate this overhead is negligible in practice.

Third, to support virtual memory for host user space applications in EL0 and the kernel running in EL2 while preserving normal Linux virtual memory management semantics, we make two Linux modifications. One provides a way to bridge the differences between the different page table formats of EL0 and EL2, and the other uses the single EL2 page table to mimic the behavior using two EL0/EL1 page tables.

Bridging the differences between different page table formats of EL0 and EL2 is important because Linux memory management is designed around the assumption that the same page tables are used from both user and kernel mode, with potentially different access permissions between the two modes. This allows Linux to maintain a consistent per-process view of virtual memory from both the kernel and user space. Violating this assumption would require invasive and complex changes to the Linux kernel. *el2*Linux takes advantage of the fact the differences between EL0/EL1 and EL2 page table formats are relatively small and can be bridged to use the same page tables for both EL0 and EL2 by slightly relaxing a security feature and accepting a higher TLB invalidation frequency on some workloads.

*el2*Linux relaxes a security feature because the EL2 page table format only has a single non-execute bit which must be shared by EL0 and EL2 to use the same page tables for both EL0 and EL2. When setting this bit on a page table entry which is used in both EL2 and EL0, the page is not executable by the kernel or user space, and when clearing this bit, the page is executable by both. Since kernel pages containing code must be executable by the kernel, the single non-execute bit means they end up executable by both user space and the kernel. This problem does not exist for EL1 page tables because they support two bits to control if a page is executable or non-executable, one for EL0 and one for EL1. We emphasize that while this is a slight relaxation of a security feature, it is not a direct security exploit. All kernel pages can still not be read or written from user space, but only executed, and can still only be executed with user privileges. This security relaxation may work against the purpose of kernel hardening techniques such as kernel address space randomization (KASLR), because user software can try to execute random addresses in the kernel's address space and rely on signals to regain control, and by observing the register state of the CPU or by observing other side effects, applications can attempt to reason about where the kernel maps its code and data within its address space.

Alternative solutions exist to support virtual memory for host user space applications in EL0 without relaxing this security feature, but require more invasive changes to Linux. One approach would be to simply not use the same page tables between the kernel and user space and maintain two page tables per process, one used by the host user space in EL0 and one used by the kernel in EL2. This solution would require additional synchronization mechanisms to make sure the two page tables always maintained a consistent view of a process address space between user space threads and the kernel. Another approach would be to not allow Linux to access user space pointers from within the kernel and instead require Linux to translate every user space virtual address into a kernel virtual address by walking the EL0 user space page tables in software from within the kernel on every user access such as read or write system calls that transfer data between user space processes and the kernel.

*el2*Linux may incur a higher TLB invalidation frequency because virtual memory accesses performed in EL2 are not tagged with an Address Space Identifier (ASID), which are used to distinguish different address space resolutions in the TLB to avoid having to invalidate TLB entries when changing address spaces, for example when switching between processes. While the kernel address space is shared for all processes, the kernel also some times accesses user space addresses when copying data between user space, for example when handling system calls. Such accesses should be tagged with the process ASID to ensure that TLB entries only match for the right process. Since memory accesses performed in EL2 are not associated with a ASID, we must invalidate all EL2 entries in the TLB when switching between processes. This does not affect TLB entries for memory accesses done by user space applications, as these still run in EL0 and all EL0 accesses still use ASIDs. We did not observe a slowdown in overall system performance as a result of this design, and estimate that for most virtualization workloads the effect will be minimal, but it could be substantial for other host workloads. Note that VHE hardware uses ASIDs in EL2 and does not have this limitation.

Finally, *el2*Linux uses an approach similar to x86 Linux to enable a single EL2 page table to mimic the behavior using two EL0/EL1 page tables. Instead of having separate page tables for user and kernel address spaces as is done in EL1, *el2*Linux splits a single address space so that half is for user space and the other half is for a shared kernel space among all processes. Similar to x86 Linux, *el2*Linux only maintains a single copy of the second level page tables for the kernel and points to these from the first level page table across all processes. ARM supports a maximum of 48 bits of contiguous virtual addresses, resulting in a maximum of 47 bits of address space for both the kernel and each user space process.

4 Hypervisor Redesign

While running the hypervisor OS kernel in a separate CPU mode is a key aspect of our approach, it turns out that this alone is insufficient to significantly improve virtualization performance, as we will show in Section 5. The hypervisor itself must also be redesigned to take advantage of not having to multiplex the same CPU mode between the hypervisor OS kernel and the VM. We redesigned KVM/ARM based on this insight. A key challenge was to do this in such a way that our modifications could be accepted by the Linux community,

which required also supporting legacy systems in which users may still choose to run the hypervisor OS kernel in EL1. We describe three techniques we used to redesign KVM/ARM's execution flow to improve performance.

First, we redesigned KVM/ARM to avoid saving and restoring EL1 registers on every transition between a VM and the hypervisor. The original KVM/ARM had to save and restore EL1 state on every transition because EL1 was shared between a VM's guest OS kernel and the hypervisor OS kernel. Since the hypervisor OS kernel now runs in EL2 and does not use the EL1 state anymore, it can load the VM's EL1 state into CPU registers when it runs the VM's virtual CPU (VCPU) on the physical CPU for the first time. It does not have to save or modify this state again until it runs another VCPU or has to configure its EL1 runtime to run applications in host user space. This entails not only eliminating copying EL1 state to in-memory hypervisor data structures on each transition between a VM and the hypervisor, but also modifying KVM/ARM to directly access the physical CPU for the running VCPU's EL1 register state since the hypervisor data structures may be out of date. To preserve backwards compatibility to also use KVM/ARM without Linux running in EL2, we keep track of whether a VCPU's EL1 registers are loaded onto the physical CPU or stored in memory and direct accesses to EL1 registers in KVM/ARM to the appropriate location using access functions.

Second, we redesigned KVM/ARM to avoid enabling and disabling virtualization features on every transition between the VM and the hypervisor. The original KVM/ARM had to disable virtualization features when running the hypervisor OS kernel so it could have full access to the underlying hardware, but then enable virtualization features when running a VM so it only had restricted access to virtualized hardware. The configuration of virtualization features such as stage 2 translations, virtual interrupts, and traps on sensitive instructions only apply to software running in EL1 and EL0. Since the hypervisor OS kernel now runs in EL2, it automatically has full access to the underlying hardware and the configuration of virtualization features do not apply to it. Instead, the virtualization features simply remain enabled for running VMs in EL1 and EL0, eliminating frequent writes to the group of special EL2 registers that configures the virtualization features. The only time the virtualization features need to be disabled is for running host user space applications and its supporting EL1 runtime, which happens relatively infrequently.

Third, we redesigned KVM/ARM to avoid the use of shared, intermediate data structures between EL1 and EL2. The original KVM/ARM using split-mode virtualization had to communicate across EL1 and EL2 modes via intermediate data structures mapped in both CPU modes because much of the hypervisor functionality was implemented in the hypervisor OS kernel running in EL1 but needed to have some aspect run in EL2 to program EL2 hardware. The hypervisor ends up processing data twice, once in EL1 which results in writing data to an intermediate data structure, and once in EL2 to process the intermediate data structure and program the hardware. Similarly, duplicative processing also happened when intermediate data structures were used to store EL2 state that needed to be read by the hypervisor OS kernel in EL1 but could only be read by the hypervisor in EL2. This complicates the code and results in many conditional statements. To make matters worse, since EL1 and EL2 run in separate address spaces, accessing the intermediate data structures can result in a TLB miss for both EL1 and EL2. Since the hypervisor OS kernel now runs in EL2 together with the rest of KVM/ARM, there is no longer any need for these intermediate data structures. The previously separate logic to interact with the rest of the hypervisor OS kernel and to program or access the EL2 hardware can be combined into a single optimized step, resulting in improved performance.

A prime example of how eliminating the need for intermediate data structures helped was the virtual interrupt controller (VGIC) implementation, which is responsible for handling virtual interrupts for VMs. VGIC hardware state is only accessible and programmable in EL2, however hypervisor functionality pertaining to virtual interrupts relies on the hypervisor OS kernel, which ran in EL1 with the original KVM/ARM. Since it was not clear when running in EL2 what VGIC state would be needed in EL1, the original KVM/ARM would conservatively copy all of the VGIC state to intermediate data structures so it was accessible in EL1, so that, for example, EL1 could save the state to in-memory data structures if it was going to run another VM. Furthermore, the original KVM/ARM would identify any pending virtual interrupts but then could only write this information to an intermediate data structure, which then needed to be later accessed in EL2 to write them into the VGIC hardware.

Since the hypervisor OS kernel now runs in EL2 together with the rest of KVM/ARM, the redesigned KVM/ARM no longer needs to conservatively copy all VGIC state to intermediate data structures, but can instead have the hypervisor kernel access VGIC state directly whenever needed. Furthermore, since the redesign simplified the execution flow, it became clear that some VGIC registers were never used by KVM and thus never needed to be copied, saved, or restored. It turns out that eliminating extra VGIC register accesses is very beneficial because VGIC register accesses are expensive. Similarly, since the hypervisor OS kernel now runs in EL2, there is no need to check for pending virtual interrupts in both EL1 and EL2. Instead these steps can be combined into a single optimized step that also writes them into the VGIC hardware as needed. As part of this redesign, it became clear that the common case that should be made fast is that there are no pending interrupts so only a single simple check should be required. We further optimized this step by avoiding the need to hold locks in the common case, which was harder to do with the original KVM/ARM code base that

had to synchronize access to intermediate data structures.

To maintain backwards compatibility support for systems not running the hypervisor and its host OS kernel in EL2, while not adding additional runtime overhead from conditionally execution almost all operations in the run loop, we take advantage of the static key infrastructure in Linux. Static keys patch the instruction flow at runtime to avoid conditional branches, and instead replaces no-ops with unconditional branches when a certain feature is enabled. During initialization of KVM/ARM, we activate or deactivate the static branch depending on whether KVM/ARM runs in EL2 or EL1. For example, the run loop uses a static branch to decide if it should call the lowvisor to start switching to a VM in EL2, or if it should simply run the VM if the hypervisor is already running in EL2.

5 Experimental Results

We have successfully merged many of our implementation changes in redesigning KVM/ARM into the mainline Linux kernel, demonstrating the viability of our approach. Getting changes accepted into mainline Linux takes time, and as such, our improvements have been merged into mainline Linux over the course of Linux kernel versions v4.5 through v4.8, with remaining changes scheduled to be applied in upcoming kernel versions.

We evaluate the performance of our new hypervisor design using both microbenchmarks and real application workloads on ARM server hardware. Since no VHE hardware is publicly available yet, we ran workloads on non-VHE ARM hardware using *el2*Linux. We expect that *el2*Linux provides a conservative but similar measure of performance to what we would expect to see with VHE since the critical hypervisor execution paths are almost identical between the two, and VHE does not introduce hardware features that would cause runtime overhead from the hardware. In this sense, these measurements provide the first quantitative evaluation of the benefits of VHE, and provide chip designers with useful experimental data to evaluate whether or not to support VHE in future silicon. We also verified the functionality and correctness of our VHE-based implementation on ARM software models supporting VHE. As a baseline for comparison, we also provide results using KVM on x86 server hardware.

ARM measurements were done using a 64-bit ARMv8 AMD Seattle (Rev.B0) server with 8 Cortex-A57 CPU cores, 16 GB of RAM, a 512 GB SATA3 HDD for storage, and a AMD 10 GbE (AMD XGBE) NIC device. For benchmarks that involve a client interfacing with the ARM server, we ran the clients on an x86 machine with 24 Intel Xeon CPU 2.20 GHz cores and 96 GB RAM. The client and the server were connected using 10 GbE and we made sure the interconnecting switch was not saturated during our measurements. x86 measurements were done using Dell PowerEdge r320 servers, each with a 64-bit Xeon 2.1 GHz E5-2450 with 8 physical CPU cores. Hyper-Threading was disabled on the r320 servers to provide a similar hardware configuration to the ARM servers. Each r320 node had 16 GB of RAM, 4 500 GB 7200 RPM SATA RAID5 HDs for storage, and a Dual-port Mellanox MX354A 10 GbE NIC. For benchmarks that involve a client interfacing with the x86 server, we ran the clients on an identical x86 client. CloudLab [10] infrastructure was used for x86 measurements, which also provides isolated 10 GbE interconnect between the client and server.

To provide comparable measurements, we kept the software environments across all hardware platforms and hypervisors the same as much as possible. KVM/ARM was configured with passthrough networking from the VM to an AMD XGBE NIC device using Linux's VFIO direct device assignment framework. KVM on x86 was configured with passthrough networking from the VM to one of the physical functions of the Mellanox MX354A NIC. Following best practices, we configured KVM virtual block storage with `cache=none`. We configured power management features on both server platforms and ensured both platforms were running at full performance. All hosts and VMs used Ubuntu 14.04 with identical software configurations. The client machine used for workloads involving a client and server used the same configuration as the host and VM, but using Ubuntu's default v3.19.0-25 Linux kernel.

We ran benchmarks on bare-metal machines and in VMs. Each physical or virtual machine instance used for running benchmarks was configured as a 4-way SMP with 12 GB of RAM to provide a common basis for comparison. This involved two configurations: (1) running natively on Linux capped at 4 cores and 12 GB RAM, (2) running in a VM using KVM with 8 physical cores and 16 GB RAM with the VM capped at 4 virtual CPUs (VCPUs) and 12 GB RAM. For network related benchmarks, the clients were run natively on Linux and configured to use the full hardware available.

To minimize measurement variability, we pinned each VCPU of the VM to a specific physical CPU (PCPU) and ensured that no other work was scheduled on that PCPU. We also statically allocated interrupts to a specific CPU, and for application workloads in VMs, the physical interrupts on the host system were assigned to a separate set of PCPUs from those running the VCPUs.

We compare across Linux v4.5 and v4.8 on ARM to quantify the impact of our improvements, as the former does not contain any of them while the latter contains a subset of our changes merged into mainline Linux. To ensure that our results are not affected by other changes to Linux between the two versions, we ran both v4.5 and v4.8 Linux natively on both the ARM and x86 systems and compared the results and we found that there were no noticeable differences between these versions of Linux. For comparison purposes, we measured four different system configurations, ARM, ARM EL2, ARM EL2 OPT, and x86. ARM uses vanilla KVM/ARM in Linux v4.5, the kernel version before any of our imple-

Name	Description
Hypercall	Transition from the VM to the hypervisor and return to the VM without doing any work in the hypervisor. Measures bidirectional base transition cost of hypervisor operations.
I/O Kernel	Trap from the VM to the emulated interrupt controller in the hypervisor OS kernel, and then return to the VM. Measures a frequent operation for many device drivers and baseline for accessing I/O devices supported by the hypervisor OS kernel.
I/O User	Trap from the VM to the emulated UART in QEMU and then return to the VM. Measures base cost of operations that access I/O devices emulated in the hypervisor OS user space.
Virtual IPI	Issue a virtual IPI from a VCPU to another VCPU running on a different PCPU, both PCPUs executing VM code. Measures time between sending the virtual IPI until the receiving VCPU handles it, a frequent operation in multi-core OSes.

Table 1: Microbenchmarks

mentation changes were merged into Linux. ARM EL2 uses the same KVM/ARM in Linux v4.5 but with modifications to run *el2*Linux to quantify the benefits of running Linux in EL2 without also redesigning the KVM/ARM hypervisor itself. ARM EL2 OPT uses our redesigned KVM/ARM in Linux v4.8, including all of the optimizations described in this paper, both those already merged into Linux v4.8 and those scheduled to be applied in upcoming Linux versions.

5.1 Microbenchmark Results

We first ran various microbenchmarks as listed in Table 1, which are part of the KVM unit test framework [23]. We slightly modified the test framework to measure the cost of virtual IPIs and to obtain cycle counts on the ARM platform to ensure detailed results by configuring the VM with direct access to the cycle counter. Table 2 shows the microbenchmark results. Measurements are shown in cycles instead of time to provide a useful comparison across server hardware with different CPU frequencies.

The Hypercall measurement quantifies the base cost of any operation where the hypervisor must service the VM. Since KVM handles hypercalls in the host OS kernel, this metric also represents the cost of transitioning between the VM and the hypervisor OS kernel. For Hypercall, the ARM EL2 OPT is a mere 12% of the ARM cost and roughly 50% of the x86 cost, measured in cycles. Comparing the ARM and ARM EL2 costs, we see that only running the hypervisor OS kernel in a separate CPU mode from the VM kernel does not by itself yield much improvement. Instead, redesigning the hypervisor to take advantage of this fact is essential to obtain a significant performace improvement as shown by the ARM EL2 OPT costs.

The I/O Kernel measurement quantifies the cost of I/O requests to devices supported by the hypervisor OS kernel. The cost consists of the base Hypercall cost plus doing some work in the hypervisor OS kernel. For I/O Kernel, the

Microbenchmark	ARM	ARM EL2	ARM EL2 OPT	x86
Hypercall	6,413	6,277	752	1,437
I/O Kernel	8,034	7,908	1,604	2,565
I/O User	10,012	10,186	7,630	6,732
Virtual IPI	13,121	12,562	2,526	3,102

Table 2: Microbenchmark Measurements (cycle counts)

ARM EL2 OPT cost is only 20% of the original ARM cost because of the significant improvement in the Hypercall cost component of the overall I/O Kernel operation.

The I/O User measurement quantifies the cost of I/O requests that are handled by host user space. For I/O User, ARM EL2 OPT cost is only reduced to 76% of the ARM cost. The improvement is less in this case because our *el2*Linux implementation requires restoring the host's EL1 state before returning to user space since running user applications in EL0 without VHE uses an EL1 runtime, as discussed in Section 3.2. However, returning to user space from executing the VM has always been known to be slow, as can also be seen with the x86 I/O User measurement in Table 2. Therefore, most hypervisor configurations do this very rarely. For example, the vhost configuration of virtio [25] paravirtualized I/O that is commonly used with KVM completely avoids going to host user space when doing I/O.

Finally, the Virtual IPI measurement quantifies the cost of issuing virtual IPIs (Inter Processor Interrupts), a frequent operation in multi-core OSes. It involves exits from both the sending VCPU and receiving VCPU. The sending VCPU exits because sending an IPI traps and is emulated by the underlying hypervisor. The receiving VCPU exits because it gets a physical interrupt which is handled by the hypervisor. For Virtual IPI, ARM EL2 OPT cost is only 19% of the original ARM cost because of the significant improvement in the Hypercall cost, which benefits both the sending and receiving VCPUs in terms of lower exit costs.

Our microbenchmark measurements show that our KVM/ARM redesign is roughly an order of magnitude faster than KVM/ARM's legacy split-mode design in transitioning between the VM and the hypervisor. The ARM EL2 numbers show slight improvement over the ARM numbers, due to the removal of the double trap cost [14] introduced by split-mode virtualization. However, a key insight based on our implementation experience and these results is that only running the hypervisor OS kernel in a separate CPU mode from the VM kernel is insufficient to have much of a performance benefit, even on architectures like ARM which have the ability to quickly switch between the two separate CPU modes without having to multiplex any state. However, if the hypervisor is designed to take advantage of running the hypervisor OS kernel in a separate mode, and the hardware provides the capabilities to do so and to switch quickly between the two modes, then the cost of low-level VM-to-hypervisor interactions can be much lower than on

Name	Description
Kernbench	Compilation of the Linux 3.17.0 kernel using the allnoconfig for ARM using GCC 4.8.2.
Hackbench	`hackbench` [24] using Unix domain sockets and 100 process groups running with 500 loops.
Netperf	`netperf` v2.6.0 [22] starting netserver on the server and running with its default parameters on the client in three modes: TCP_STREAM, TCP_MAERTS, and TCP_RR, measuring throughput and latency, respectively.
Apache	`Apache` v2.4.7 Web server with a remote client running `ApacheBench` [28] v2.3, which measures number of handled requests per second serving the 41 KB index file of the GCC 4.4 manual using 100 concurrent requests.
Memcached	`memcached` v1.4.14 using the `memtier` benchmark v1.2.3 with its default parameters.

Table 3: Application Benchmarks

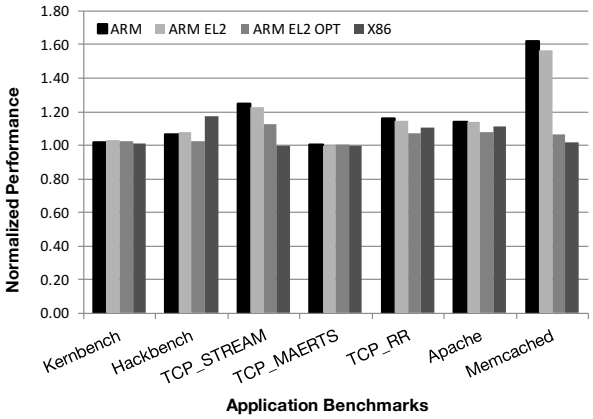


Figure 2: Application Benchmark Performance

systems like x86, even though they have highly optimized VM Entry and Exit hardware mechanisms to multiplex a single CPU mode between the hypervisor and the VM.

5.2 Application Benchmark Results

We next ran a mix of widely-used CPU and I/O intensive application workloads as listed in Table 3. For workloads involving a client and a server, we ran the client on a dedicated machine and the server on the configuration being measured, ensuring that the client was never saturated during any of our experiments. Figure 2 shows the relative performance overhead of executing in a VM compared to natively without virtualization.

We normalize measurements to native execution for the respective platform, with one being the same as native performance. ARM numbers are normalized to native execution on the ARM platform, and x86 numbers are normalized to native execution on the x86 platform. Lower numbers mean less overhead and therefore better overall performance. We focus on normalized overhead as opposed to absolute performance since our goal is to improve VM performance by reducing the overhead from intervention of the hypervisor and from switching between the VM and the hypervisor OS kernel.

Like the microbenchmark measurements in Section 5.1, the application workload measurements show that ARM EL2 performs similarly to ARM across all workloads, showing that running the hypervisor OS kernel in a separate CPU mode from the VM kernel without changing the hypervisor does not benefit performance much. The ARM EL2 OPT results, however, show significant improvements across a wide range of applications workloads.

For cases in which original ARM did not have much overhead, ARM EL2 OPT performs similarly to original ARM as there was little room for improvement. For example, Kernbench runs mostly in user mode in the VM and seldom traps to the hypervisor, resulting in very low overhead on both ARM and x86. However, the greater the initial overhead for original ARM, the greater the performance improvement achieved with ARM EL2 OPT. For example, original ARM incurs more than 60% overhead for Memcached while ARM EL2 OPT reduces that overhead by more than five times to roughly 10% compared to native execution. Memcached causes frequent traps to the hypervisor OS kernel to process, configure, and forward physical interrupts. As a result, this workload benefits greatly from the much reduced hypercall cost for ARM EL2 OPT compared to original ARM. As another example, original ARM incurs roughly 15% overhead for Apache while ARM EL2 OPT reduces that overhead by roughly 50% to 8% compared to native execution, which is even smaller than x86. Apache requires processing network interrupts and sending virtual IPIs, both of which benefit from the reduced hypercall cost for ARM EL2 OPT.

It is instructive to take a closer look at the various Netperf measurements, TCP_STREAM, TCP_RR and TCP_MAERTS, which show ARM EL2 OPT providing different performance improvements over original ARM and x86. Since we use passthrough to directly assign the network device to the VM, the primary source of overhead comes from interrupt handling because the VM can otherwise directly program the device without intervention from the hypervisor. The network devices used on both the ARM and x86 servers generate physical RX interrupts when receiving network data, which is the primary operation of TCP_STREAM and TCP_RR. These physical interrupts are handled by VFIO in the host kernel and KVM must forward them as virtual interrupts to the VM, which results in execution overhead. The driver for the AMD XGBE NIC used in the ARM server frequently masks and unmasks interrupts for this device due to driver implementation details and support for NAPI, which switches between interrupt driven and polling mode for the VM network driver. On the other hand, the driver for the Mellanox NIC used in the x86 server does not enable and disable IRQs using the interrupt controller, but instead manages masking of interrupts at the device level, which avoids traps to the hypervisor for these operations because the device is directly assigned to the VM.

TCP_STREAM is a throughput benchmark and since

x86 has fewer traps to the hypervisor than ARM due to these NIC diferences, x86 has lower virtualization overhead than any ARM configuration, including ARM EL2 OPT. The same explanation applies to Memcached as well. The TCP_RR workload is a latency measurement benchmark, which sends a single network packet back and forward between the client and the server in serial, and every single packet causes an interrupt for both ARM and x86, resulting in overhead on both platforms. Since ARM EL2 OPT has lower transition costs between the VM and hypervisor when comparing against either original ARM or x86, it also ends up having the lowest overhead for TCP_RR. For both TCP_STREAM and TCP_RR, ARM EL2 OPT reduces the overhead of original ARM by approximately 50% as a result of the reduced cost of transitioning between the hypervisor OS kernel and the VM when masking and unmasking virtual interrupts, and when forwarding physical interrupts as virtual interrupts, respectively. TCP_MAERTS shows almost no overhead for all configurations, because sending packets from the VM to the client generates almost no interrupts and the VMs can access the devices directly because of their passthrough device configuration.

6 Related Work

Virtualization on x86 started with software-only approaches [3, 7], but as virtualization became increasingly important, Intel introduced VMX hardware virtualization support to run VMs with unmodified guest OSes and eliminate the need for binary translation and CPU paravirtualization [29]. Initially, hypervisors using hardware virtualization support did not provide good performance [1], but as the hardware support matured and provided additional features like EPT, performance improved using VMX. Much other work has been also done on analyzing and improving the performance of x86 virtualization [27, 26, 2, 18, 9, 16, 19, 6], but none of these techniques addressed the core issue of the cost of sharing kernel mode across guest and hypervisor OS kernels.

Full-system virtualization of the ARM architecture in some ways mirrors the evolution of x86 virtualization. Early approaches were software only, could not run unmodified guest OSes, and often suffered from poor performance [20, 13, 15, 4]. As virtualization became increasingly important on ARM, ARM introduced hardware virtualization support to run VMs with unmodified guest OSes, but ARM took a very different approach to CPU virtualization that made it difficult to implement popular hypervisors such as KVM due to mismatches between the hardware support and assumptions about the software design [14]. As a result, ARM hypervisor implementations have much higher costs for many common VM operations than their x86 counterparts [12]. We show that by taking advantage of the additional CPU mode provided by ARM VE to run not only the hypervisor but also its OS kernel, in conjunction with a redesign of the hypervisor itself, it is possible to achieve superior VM performance on ARM versus x86.

7 Conclusions

Although ARM and x86 architectural support for virtualization are quite different, previous hypervisors across both architectures shared a common limitation; the need to share kernel mode state between the host OS kernel used by the hypervisor and guest OS kernels used in VMs. Our work shows for the first time how, with the proper architectural support and hypervisor design, the hypervisor and its OS kernel can be run in a separate CPU mode from VMs, avoiding the cost of multiplexing shared CPU state between the hypervisor and VMs. We show how this codesign of hardware and software support for virtualization can be used to reimplement an existing hypervisor, KVM/ARM, with evolutionary changes to the code base without requiring a clean slate implementation. This approach was essential in allowing us to merge many of our changes into mainline Linux. We show that our approach can be implemented using currently available ARM hardware, and that new hardware features in future ARM architecture versions can be used to support this approach without any changes to Linux other than to KVM/ARM itself. We show that our KVM/ARM redesign can provide an order of magnitude performance improvement over previous versions of KVM/ARM on key hypervisor operations. We also show that the combination of hardware and software virtualization support on ARM can provide roughly two times better performance than its counterpart on x86. Our results indicate that running the hypervisor and its hypervisor OS kernel in a separate CPU mode from the VMs as possible on ARM can provide superior performance to x86 approaches because it allows for faster transitions between the hypervisor and VMs. As virtualization continues to be of importance, our work provides an important counterpoint to x86 practices which we believe is instrumental in designing future virtualization support for new architectures.

8 Acknowledgments

Marc Zyngier implemented some VGIC optimizations and supported our efforts to upstream our improvements to KVM/ARM. Ard Biesheuvel helped us understand the virtual memory changes needed to run Linux in EL2. Eric Auger implemented VFIO passthrough support for ARM and provided help in configuring passthrough on ARM server hardware. Paolo Bonzini and Alex Williamson helped analyze KVM x86 performance. Mike Hibler provided support for system configurations in CloudLab. This work was supported in part by Huawei Technologies and NSF grants CNS-1422909, CNS-1563555, and CCF-1162021.

References

[1] ADAMS, K., AND AGESEN, O. 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* (Oct. 2006), pp. 2–13.

[2] AGESEN, O., MATTSON, J., RUGINA, R., AND SHELDON, J. Software Techniques for Avoiding Hardware Virtualization Exits. In *Proceedings of the 2012 USENIX Annual Technical Conference* (June 2012), pp. 373–385.

[3] 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* (Oct. 2003), pp. 164–177.

[4] BARR, K., BUNGALE, P., DEASY, S., GYURIS, V., HUNG, P., NEWELL, C., TUCH, H., AND ZOPPIS, B. The VMware Mobile Virtualization Platform: is that a hypervisor in your pocket? *SIGOPS Operating Systems Review 44*, 4 (Dec. 2010), 124–135.

[5] BONZINI, P. Virtualizing the locomotive, Sept. 2005. `https://lwn.net/Articles/657282/`.

[6] BUELL, J., HECHT, D., HEO, J., SALADI, K., AND TAHERI, H. R. Methodology for Performance Analysis of VMware vSphere under Tier-1 Applications. *VMware Technical Journal 2*, 1 (June 2013), 19–28.

[7] BUGNION, E., DEVINE, S., ROSENBLUM, M., SUGERMAN, J., AND WANG, E. Y. Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. *ACM Transactions on Computer Systems 30*, 4 (Nov. 2012), 12:1–12:51.

[8] BUGNION, E., NIEH, J., AND TSAFRIR, D. *Hardware and Software Support for Virtualization*, vol. 12 of *Synthesis Lectures on Computer Architecture*. Morgan & Claypool Publishers, Feb. 2017.

[9] CHERKASOVA, L., AND GARDNER, R. Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor. In *Proceedings of the 2005 USENIX Annual Technical Conference* (May 2005), pp. 387–390.

[10] CLOUDLAB. `http://www.cloudlab.us`.

[11] DALL, C., AND LI, S.-W. *el2*linux. `https://github.com/chazy/el2linux`.

[12] DALL, C., LI, S.-W., LIM, J. T., NIEH, J., AND KOLOVENTZOS, G. ARM Virtualization: Performance and Architectural Implications. In *Proceedings of the 43rd International Symposium on Computer Architecture* (2016), pp. 304–316.

[13] DALL, C., AND NIEH, J. KVM for ARM. In *Proceedings of the Ottawa Linux Symposium* (July 2010), pp. 45–56.

[14] DALL, C., AND NIEH, J. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Mar. 2014), pp. 333–348.

[15] DING, J.-H., LIN, C.-J., CHANG, P.-H., TSANG, C.-H., HSU, W.-C., AND CHUNG, Y.-C. ARMvisor: System Virtualization for ARM. In *Proceedings of the Ottawa Linux Symposium* (July 2012), pp. 93–107.

[16] GAMAGE, K., AND KOMPELLA, X. Opportunistic Flooding to Improve TCP Transmit Performance in Virtualized Clouds. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (Oct. 2011), pp. 24:1–24:14.

[17] GOLDBERG, R. P. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, Cambridge, MA, 1972.

[18] 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. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems* (Feb. 2012), pp. 411–422.

[19] HEO, J., AND TAHERI, R. Virtualizing Latency-Sensitive Applications: Where Does the Overhead Come From? *VMware Technical Journal 2*, 2 (Dec. 2013), 21–30.

[20] HWANG, J., SUH, S., HEO, S., PARK, C., RYU, J., PARK, S., AND KIM, C. Xen on ARM: System Virtualization using Xen Hypervisor for ARM-based Secure Mobile Phones. In *Proceedings of the 5th Consumer Communications and Newtork Conference* (Jan. 2008), pp. 257–261.

[21] INTEL CORPORATION. Intel 64 and IA-32 Architectures Software Developer's Manual, 325384-056US, Sept. 2015.

[22] JONES, R. Netperf. `http://www.netperf.org/netperf/`.

[23] KIVITY, A. KVM Unit Tests. http://www.linux-kvm.org/page/KVM-unit-tests.

[24] RED HAT. Hackbench, Jan. 2008. http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c.

[25] RUSSELL, R. virtio: Towards a De-Facto Standard for Virtual I/O Devices. *SIGOPS Operating Systems Review 42*, 5 (July 2008), 95–103.

[26] SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G. J., AND PRATT, I. Bridging the Gap between Software and Hardware Techniques for I/O Virtualization. In *Proceedings of the 2008 USENIX Annual Technical Conference* (June 2008), pp. 29–42.

[27] SUGERMAN, J., VENKITACHALAM, G., AND LIM, B.-H. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference* (June 2001), pp. 1–14.

[28] THE APACHE SOFTWARE FOUNDATION. ab. http://httpd.apache.org/docs/2.4/programs/ab.html.

[29] UHLIG, R., NEIGER, G., RODGERS, D., SANTONI, A. L., MARTINS, F. C. M., ANDERSON, A. V., BENNETT, S. M., KAGI, A., LEUNG, F. H., AND SMITH, L. Intel Virtualization Technology. *IEEE Computer 38*, 5 (May 2005), 48–56.

[30] XEN ARM WITH VIRTUALIZATION EXTENSIONS. http://wiki.xenproject.org/wiki/Xen_ARM_with_Virtualization_Extensions.

Multi-hypervisor Virtual Machines: Enabling an Ecosystem of Hypervisor-level Services

Kartik Gopalan, Rohith Kugve, Hardik Bagdi, Yaohui Hu
Computer Science, Binghamton University

Dan Williams, Nilton Bila
IBM T.J. Watson Research Center

Abstract

Public cloud software marketplaces already offer users a wealth of choice in operating systems, database management systems, financial software, and virtual networking, all deployable and configurable at the click of a button. Unfortunately, this level of customization has not extended to emerging hypervisor-level services, partly because traditional virtual machines (VMs) are fully controlled by only one hypervisor at a time. Currently, a VM in a cloud platform cannot concurrently use hypervisor-level services from multiple third-parties in a compartmentalized manner. We propose the notion of a *multi-hypervisor VM*, which is an unmodified guest that can simultaneously use services from multiple coresident, but isolated, hypervisors. We present a new virtualization architecture, called *Span virtualization*, that leverages nesting to allow multiple hypervisors to concurrently control a guest's memory, virtual CPU, and I/O resources. Our prototype of Span virtualization on the KVM/QEMU platform enables a guest to use services such as introspection, network monitoring, guest mirroring, and hypervisor refresh, with performance comparable to traditional nested VMs.

1 Introduction

In recent years, a number of hypervisor-level services have been proposed such as rootkit detection [61], live patching [19], intrusion detection [27], high availability [24], and virtual machine (VM) introspection [30, 53, 26, 49, 42, 60]. By running inside the hypervisor instead of the guest, these services can operate on multiple guests, while remaining transparent to the guests. Recent years have also seen a rise in the number of specialized hypervisors that are tailored to provide VMs with specific services. For instance, McAfee Deep Defender [46] uses a micro-hypervisor called DeepSafe to improve guest security. SecVisor [56] provides code integrity for commodity guests. CloudVisor [68] guarantees guest privacy and integrity on untrusted clouds. RTS provides a Real-time Embedded Hypervisor [52] for real-time guests. These specialized hypervisors may not provide guests with the full slate of memory, virtual CPU (VCPU), and I/O management, but rely upon either another commodity hypervisor, or the guest itself, to fill in the missing services.

Currently there is no good way to expose multiple services to a guest. For a guest which needs *multiple* hypervisor-level services, the first option is for the single controlling hypervisor to bundle all services in its supervisor mode. Unfortunately, this approach leads to a "fat" feature-filled hypervisor that may no longer be trustworthy because it runs too many untrusted or mutually distrusting services. One could de-privilege some services to the hypervisor's user space as processes that control the guest indirectly via event interposition and system calls [63, 40]. However, public cloud providers would be reluctant to execute untrusted third-party services in the hypervisor's native user space due to a potentially large user-kernel interface.

The next option is to de-privilege the services further, running each in a *Service VM* with a full-fledged OS. For instance, rather than running a single Domain0 VM running Linux that bundles services for all guests, Xen [4] can use several disaggregated [23] service domains for resilience. Service domains, while currently trusted by Xen, could be adapted to run third-party untrusted services. A service VM has a less powerful interface to the hypervisor than a user space service. However, neither user space services nor Service VMs allow control over low-level guest resources, such as page mappings or VCPU scheduling, which require hypervisor privileges.

One could use *nested virtualization* [34, 10, 48, 29] to *vertically stack* hypervisor-level services, such that a trusted base hypervisor at layer-0 (L0) controls the physical hardware and runs a service hypervisor at layer-1 (L1), which fully or partially controls the guest at layer-2

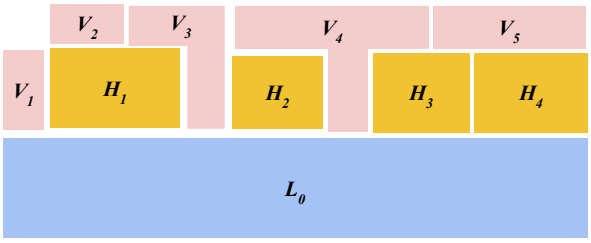


Figure 1: Span virtualization: V_1 is a traditional single-level VM. V_2 is a traditional nested VM. V_3, V_4 and V_5 are multi-hypervisor VMs.

(L2). Nested virtualization is experiencing considerable interest [25, 31, 64, 68, 55, 53, 56, 39, 7, 65, 45]. For example, one can use nesting [21] to run McAfee Deep Defender [46], which does not provide full system and I/O virtualization, as a guest on XenDesktop [20], a full commodity hypervisor, so that guests can use the services of both. Similarly, Bromium [15] uses nesting on a Xen-based hypervisor for security. Ravello [2] and XenBlanket [66, 57] use nesting on public clouds for cross-cloud portability. However, vertical stacking reduces the degree of guest control and visibility to lower layers compared to the layer directly controlling the guest. Also, the overhead of nested virtualization beyond two layers can become rather high [10].

Instead, we propose *Span virtualization*, which provides *horizontal* layering of multiple hypervisor-level services. A *Span VM*, or a multi-hypervisor VM, is an unmodified guest whose resources (virtual memory, CPU, and I/O) can be *simultaneously* controlled by multiple coresident, but isolated, hypervisors. A base hypervisor at L0 provides a core set of services and uses nested virtualization to run multiple deprivileged service hypervisors at L1. Each L1 augments L0's services by adding/replacing one or more services. Since the L0 no longer needs to implement every conceivable service, L0's footprint can be smaller than a feature-filled hypervisor. Henceforth, we use the following terms:

- **Guest** or **VM** refers to a top-level VM, with qualifiers *single-level*, *nested*, and *Span* as needed.
- **L1** refers to a service hypervisor at layer-1.
- **L0** refers to the base hypervisor at layer-0.
- **Hypervisor** refers to the role of either L0 or any L1 in managing guest resources.

Figure 1 illustrates possible Span VM configurations. One L0 hypervisor runs multiple L1 hypervisors (H_1, H_2, H_3, and H_4) and multiple guests (V_1, V_2, V_3, V_4 and V_5). V_1 is a traditional single-level (non-nested) guest that runs on L0. V_2 is a traditional nested guest that runs on only one hypervisor (H_1). The rest are multi-hypervisor VMs. V_3 runs on two hypervisors (L0 and H_1). V_4 runs on three hypervisors (L0, H_2, and H_3). V_5 is a fully nested Span VM that runs on two L1s (H_3 and H_4). This paper makes the following contributions:

- We examine the solution space for providing multiple services to a common guest and identify the relative merits of possible solutions.
- We present the design of Span virtualization which enables multiple L1s to concurrently control an unmodified guest's memory, VCPU, and I/O devices using a relatively simple interface with L0.
- We describe our implementation of Span virtualization by extending the nested virtualization support in KVM/QEMU [40] and show that Span virtualization can be implemented within existing hypervisors with moderate changes.
- We evaluate Span VMs running unmodified Linux and simultaneously using multiple L1 services including VM introspection, network monitoring, guest mirroring, and hypervisor refresh. We find that Span VMs perform comparably with nested VMs and within 0–20% of single-level VMs, across different configurations and benchmarks.

2 Solution Space

Table 1 compares possible solutions for providing multiple services to a guest. These are single-level virtualization, user space services, service VMs, nested virtualization, and Span.

First, like single-level and nested alternatives, Span virtualization provides L1s with control over the virtualized instruction set architecture (ISA) of the guest, which includes trapped instructions, memory mappings, VCPU scheduling, and I/O.

Unlike all alternatives, Span L1s support both full and partial guest control. Span L1s can range from full hypervisors that control all guest resources to specialized hypervisors that control only some guest resources.

Next, consider the impact of service failures. In single-level virtualization, failure of a privileged service impacts the L0 hypervisor, all coresident services, and all guests. For all other cases, the L0 hypervisor is protected from service failures because services are deprivileged. Furthermore, failure of a deprivileged service impacts only those guests to which the service is attached.

Next, consider the failure impact on coresident services. User space services are isolated by process-level isolation and hence protected from each other's failure. However, process-level isolation is only as strong as the user-level privileges with which the services run. Nested virtualization provides only one deprivileged service compartment. Hence services for the same guest must reside together in an L1, either in its user space or kernel. A service failure in a nested L1's kernel impacts all coresident services whereas a failure in its user

	Level of Guest Control		Impact of Service Failure			Additional Performance Overheads
	Virtualized ISA	Partial or Full	L0	Coresident Services	Guests	
Single-level	Yes	Full	Fails	Fail	All	None
User space	No	Partial	Protected	Protected	Attached	Process switching
Service VM	No	Partial	Protected	Protected	Attached	VM switching
Nested	Yes	Full	Protected	Protected in L1 user space	Attached	L1 switching + nesting
Span	Yes	Both	Protected	Protected	Attached	L1 switching + nesting

Table 1: Alternatives for providing multiple services to a common guest, assuming one service per user space process, service VM, or Span L1.

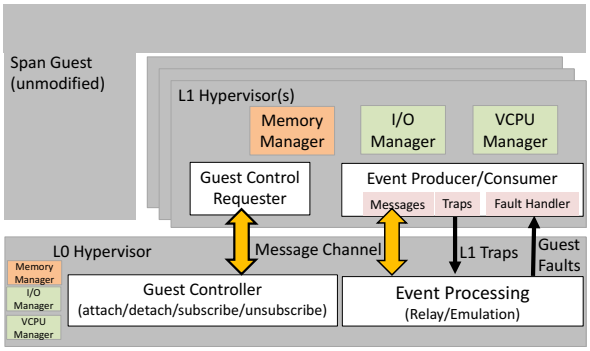


Figure 2: High-level architecture for Span virtualization.

space does not. Service VMs and Span virtualization isolate coresident services in individual VM-level compartments. Thus, failure of a service VM or Span L1 does not affect coresident services.

Finally, consider additional performance overhead over the single-level case. User space services introduce context switching overhead among processes. Service VMs introduce VM context switching overhead, which is more expensive. Nesting adds the overhead of emulating privileged guest operations in L1. Span virtualization uses nesting but supports partial guest control by L1s. Hence, nesting overhead applies only to the guest resources that an L1 controls.

3 Overview of Span Virtualization

The key design requirement for Span VMs is *transparency*. The guest OS and its applications should remain unmodified and oblivious to being simultaneously controlled by multiple hypervisors, which includes L0 and any attached L1s. Hence the guest sees a virtual resource abstraction that is indistinguishable from that of a traditional (single) hypervisor. For control of individual resources, we translate this requirement as follows.

- **Memory:** All hypervisors must have the same consistent view of the guest memory.
- **VCPUs:** All guest VCPUs must be controlled by one hypervisor at a given instant.
- **I/O Devices:** Different virtual I/O devices of the same guest may be controlled exclusively by different hypervisors at a given instant.
- **Control Transfer:** Control of guest VCPUs and/or virtual I/O devices can be transferred from one hypervisor to another, but only via L0.

Figure 2 shows the high-level architecture. A Span guest begins as a single-level VM on L0. One or more L1s can then attach to one or more guest resources and optionally subscribe with L0 for specific guest events.

Guest Control Operations: The Guest Controller in L0 supervises control over a guest by multiple L1s through the following operations.

- `[attach L1, Guest, Resource]`: Gives L1 control over the Resource in Guest. Resources include guest memory, VCPU, and I/O devices. Control over memory is shared among multiple attached L1s, whereas control over guest VCPUs and virtual I/O devices is exclusive to an attached L1. Attaching to guest VCPUs or I/O device resources requires attaching to the guest memory resource.
- `[detach L1, Guest, Resource]`: Releases L1's control over Resource in Guest. Detaching from the guest memory resource requires detaching from guest VCPUs and I/O devices.
- `[subscribe L1, Guest, Event, <GFN Range>]` Registers L1 with L0 to receive Event from Guest. The GFN Range option specifies the range of frames in the guest address space on which to track the memory event. Presently we support only memory event subscription. Other guest events of interest could include SYSENTER instructions, port-mapped I/O, etc.
- `[unsubscribe L1, Guest, Event, <GFN Range>]` Unsubscribes L1 Guest Event.

The Guest Controller also uses administrative policies to resolve apriori any potential conflicts over a guest control by multiple L1s. While this paper focuses on mechanisms rather than specific policies, we note that the problem of conflict resolution among services is not unique to Span. Alternative techniques also need ways to prevent conflicting services from controlling the same guest.

Isolation and Communication: Another design goal is to compartmentalize L1 services, from each other and from L0. First, L1s must have lower execution privilege compared to L0. Secondly, L1s must remain isolated from each other. These two goals are achieved by deprivileging L1s using nested virtualization and executing them as separate guests on L0. Finally, L1s must remain unaware of each other during execution. This goal is achieved by requiring L1s to receive any subscribed guest events that are generated on other L1s only via L0.

There are two ways that L0 communicates with L1s: implicitly via *traps* and explicitly via *messages*. Traps allow L0 to transparently intercept certain memory management operations by L1 on the guest. Explicit messages allow an L1 to directly request guest control from L0. An *Event Processing* module in L0 traps runtime updates to guest memory mappings by any L1 and synchronizes guest mappings across different L1s. The event processing module also relays guest memory faults that need to be handled by L1. A bidirectional *Message Channel* relays explicit messages between L0 and L1s including attach/detach requests, memory event subscription/notification, guest I/O requests, and virtual interrupts. Some explicit messages, such as guest I/O requests and virtual interrupts, could be replaced with implicit traps. Our choice of which to use is largely based on ease of implementation on a case-by-case basis.

Continuous vs. Transient Control: Span virtualization allows L1's control over guest resources to be either *continuous* or *transient*. Continuous control means that an L1 exerts uninterrupted control over one or more guest resources for an extended period of time. For example, an intrusion detection service in L1 that must monitor guest system calls, VM exits, or network traffic, would require continuous control of guest memory, VCPUs, and network device. Transient control means that an L1 acquires full control over guest resources for a brief duration, provides a short service to the guest, and releases guest control back to L0. For instance, an L1 that periodically checkpoints the guest would need transient control of guest memory, VCPUs, and I/O devices.

4 Memory Management

A Span VM has a single guest physical address space which is mapped into the address space of all attached L1s. Thus any memory write on a guest page is immediately visible to all hypervisors controlling the guest. Note that all L1s have the same visibility into the guest memory due to the horizontal layering of Span virtualization, unlike the vertical stacking of nested virtualization, which somewhat obscures the guest to lower layers.

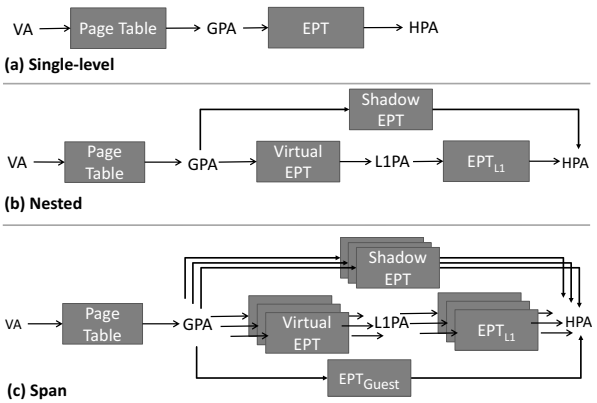


Figure 3: Memory translation for single-level, nested, and Span VMs. VA = Virtual Address; GPA = Guest Physical Address; L1PA = L1 Physical Address; HPA = Host Physical Address.

4.1 Traditional Memory Translation

In modern x86 processors, hypervisors manage the physical memory that a guest can access using a virtualization feature called *Extended Page Tables* (EPT) [37], also called Nested Page Tables in AMD-V [5].

Single-level virtualization: Figure 3(a) shows that for single-level virtualization, the guest page tables map virtual addresses to guest physical addresses (VA to GPA in the figure). The hypervisor uses an EPT to map guest physical addresses to host physical addresses (GPA to HPA). Guest memory permissions are controlled by the combination of permissions in guest page table and EPT.

Whenever the guest attempts to access a page that is either not present or protected in the EPT, the hardware generates an *EPT fault* and traps into the hypervisor, which handles the fault by mapping a new page, emulating an instruction, or taking other actions. On the other hand, the hypervisor grants complete control over the traditional paging hardware to the guest. A guest OS is free to maintain the mappings between its virtual and guest physical address space and update them as it sees fit, without trapping into the hypervisor.

Nested virtualization: Figure 3(b) shows that for nested virtualization, the guest is similarly granted control over the traditional paging hardware to map virtual addresses to its guest physical address space. L1 maintains a *Virtual EPT* to map the guest pages to pages in L1's physical addresses space, or L1 pages. Finally, one more translation is required: L0 maintains EPT_{L1} to map L1 pages to physical pages. However, x86 processors can translate only two levels of addresses in hardware, from guest virtual to guest physical to host physical address. Hence the Virtual EPT maintained by L1 needs to be *shadowed* by L0, meaning that the Virtual EPT and EPT_{L1} must be compacted by L0 during runtime into a

Shadow EPT that directly maps guest pages to physical pages. To accomplish this, manipulations to the Virtual EPT by L1 trigger traps to L0. Whenever L1 loads a Virtual EPT, L0 receives a trap and activates the appropriate Shadow EPT. This style of nested page table management is also called *multi-dimensional paging* [10].

EPT faults on guest memory can be due to (a) the guest accessing its own pages that have invalid Shadow EPT entries, and (b) the L1 directly accessing guest pages that have invalid EPT_{L1} entries to perform tasks such as I/O processing and VM introspection (VMI). Both kinds of EPT faults are first intercepted by L0. L0 examines a Shadow EPT fault to further determine whether it is due to a invalid Virtual EPT entry; such faults are forwarded to L1 for processing. Otherwise, faults due to invalid EPT_{L1} entries are handled by L0.

Finally, an L1 may modify the Virtual EPT it maintains for a guest in the course of performing its own memory management. However, since the Virtual EPT is shadowed by L0, all Virtual EPT modifications cause traps to L0 for validation and a Shadow EPT update.

4.2 Memory Translation for Span VMs

In Span virtualization, L0 extends nested EPT management to guests that are controlled by multiple hypervisors. Figure 3(c) shows that a Span guest has multiple Virtual EPTs, one per L1 that is attached to the guest. When an L1 acquires control over a guest's VCPUs, the L0 shadows the guest's Virtual EPT in the L1 to construct the corresponding Shadow EPT, which is used for memory translations. In addition, an EPT_{Guest} is maintained by L0 for direct guest execution on L0. A guest's memory mappings in Shadow EPTs, the EPT_{Guest}, and the EPT_{L1} are kept synchronized by L0 upon page faults so that every attached hypervisor sees a consistent view of guest memory. Thus, a guest virtual address leads to the same host physical address irrespective of the Shadow EPT used for the translation.

4.3 Memory Attach and Detach

A Span VM is initially created directly on L0 as a single-level guest for which the L0 constructs a regular EPT. To attach to the guest memory, a new L1 requests L0, via a hypercall, to map guest pages into its address space.

Figure 4 illustrates that L1 reserves a range in the L1 physical address space for guest memory and then informs L0 of this range. Next, L1 constructs a Virtual EPT for the guest which is shadowed by L0, as in the nested case. Note that the reservation in L1 physical address space does not immediately allocate physical memory. Rather, physical memory is allocated lazily upon guest memory faults. L0 dynamically populates the reserved address range in L1 by adjusting the mappings in EPT_{L1}

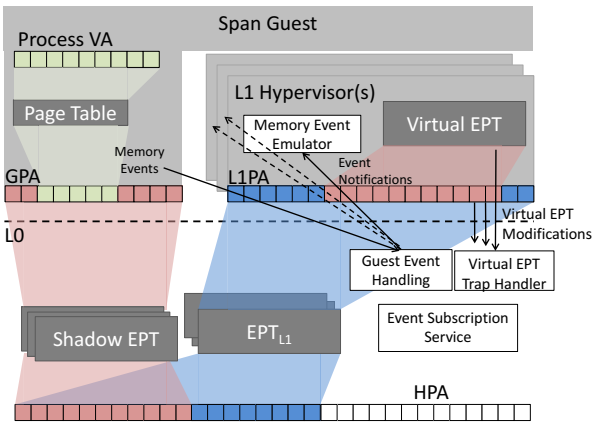


Figure 4: Span memory management overview.

and the Shadow EPT. A memory-detach operation correspondingly undoes the EPT_{L1} mappings for guest and releases the reserved L1 address range.

4.4 Synchronizing Guest Memory Maps

To enforce a consistent view of guest memory across all L1s, L0 synchronizes memory mappings upon two events: EPT faults and Virtual EPT modifications.

Fault handling for Span VMs extends the corresponding mechanism for nested VMs described earlier in Section 4.1. The key difference in the Span case is that L0 first checks if a host physical page has already been mapped to the faulting guest page. If so, the existing physical page mapping is used to resolve the fault, else a new physical page is allocated.

As with the nested case, modifications by an L1 to establish Virtual EPT mappings trap to a Virtual EPT trap handler in L0, shown in Figure 4. When the handler receives a trap due to a protection modification, it updates each corresponding EPT_{L1} with the new least-permissive combination of page protection. Our current prototype allows protection modifications but disallows changes to established GPA-to-L1PA mappings to avoid having to change mappings in multiple EPTs.

4.5 Memory Event Subscription

An L1 attached to a guest may wish to monitor and control certain memory-related events of the guest to provide a service. For instance, an L1 that provides live checkpointing or guest mirroring may need to perform *dirty page tracking* in which pages to which the guest writes are periodically recorded so they can be incrementally copied. As another example, an L1 performing intrusion detection using VM introspection might wish to monitor a guest's attempts to execute code from certain pages.

In Span virtualization, since multiple L1s can be attached to a guest, the L1 controlling the guest's VCPUs may differ from the L1s requiring the memory event notification. Hence L0 provides a *Memory Event Subscrip-*

tion interface to enable L1s to independently subscribe to guest memory events. An L1 subscribes with L0, via the message channel, requesting notifications when a specific type of event occurs on certain pages of a given guest. When the L0 intercepts the subscribed events, it notifies all L1 subscribers via the message channel. Upon receiving the event notification, a memory event emulator in each L1, shown in Figure 4, processes the event and responds back to L0, either allowing or disallowing the guest's memory access which triggered the event. The response from the L1 also specifies whether to maintain or discontinue the L1's event subscription on the guest page. For example, upon receiving a write event notification, an L1 that performs dirty page tracking will instruct L0 to allow the guest to write to the page, and cancel the subscription for future write events on the page, since the page has been recorded as being dirty. On the other hand, an intrusion detection service in L1 might disallow write events on guest pages containing kernel code and maintain future subscription. L0 concurrently delivers event notifications to all L1 subscribers. Guest memory access is allowed to proceed only if all subscribed L1s allow the event in their responses.

To intercept a subscribed memory event on a page, the L0 applies the event's mask to the corresponding EPT_{L1} entry of each L1 attached to the guest. Updating EPT_{L1} prompts L0 to update the guest's Shadow EPT entry with the mask, to capture guest-triggered memory events. Updating EPT_{L1} entries also captures the events resulting from direct accesses to guest memory by an L1 instead of the guest. For instance, to track write events on a guest page, the EPT entry could be marked read-only after saving the original permissions for later restoration.

5 I/O Control

In this work, guests use paravirtual devices [54, 6] which provide better performance than device emulation [59] and provide greater physical device sharing among guests than direct device assignment [11, 12, 50].

For single-level virtualization, the guest OS runs a set of paravirtual frontend drivers, one for each virtual device, including block and network devices. The hypervisor runs the corresponding backend driver. The frontend and the backend drivers communicate via a shared ring buffer to issue I/O requests and receive responses. The frontend places an I/O request in the ring buffer and notifies the backend through a *kick* event, which triggers a VM exit to the hypervisor. The backend removes the I/O request from the ring buffer, completes the request, places the I/O response in the ring buffer, and injects an I/O completion interrupt to the guest. The interrupt handler in the frontend then picks up the I/O response from the ring buffer for processing. For nested guests, paravirtual drivers are used at both levels.

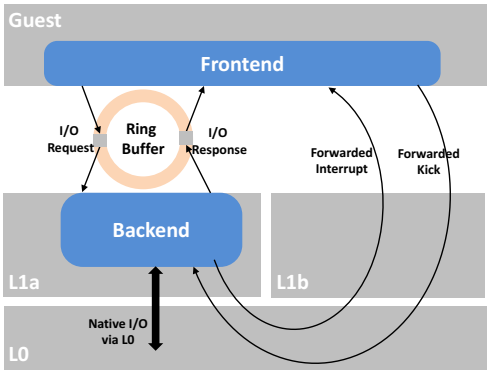


Figure 5: Paravirtual I/O for Span VMs. L1a controls the guest I/O device and L1b controls the VCPUs. Kicks from L1b and interrupts from L1a are forwarded via L0.

For Span guests, different L1s may control guest VCPUs and I/O devices. If the same L1 controls both guest VCPUs and the device backend then I/O processing proceeds as in the nested case. Figure 5 illustrates the other case, when different L1s control guest VCPUs and backends. L1a controls the backend and L1b controls the guest VCPUs. The frontend in the guest and backend in L1a exchange I/O requests and responses via the ring buffer. However, I/O kicks are generated by guest VCPUs controlled by L1b, which forward the kicks to L1a. Likewise, L1a forwards any virtual interrupts from the backend to L1b, which injects the interrupt to the guest VCPUs. Kicks from the frontend and virtual interrupts from the backend are forwarded between L1s via L0 using the message channel.

6 VCPU Control

In single-level virtualization, L0 controls the scheduling of guest VCPUs. In nested virtualization, L0 delegates guest VCPU scheduling to an L1. The L1 schedules guest VCPUs on its own VCPUs and L0 schedules the L1's VCPUs on PCPUs. This hierarchical scheduling provides the L1 some degree of control over customized scheduling for its guests.

Span virtualization can leverage either single-level or nested VCPU scheduling depending on whether the L0 or an L1 controls a guest's VCPUs. Our current design requires that all VCPUs of a guest be controlled by one of the hypervisors at any instant. However, control over guest VCPUs can be transferred between hypervisors if needed. When L0 initiates a Span VM, it initializes the all the VCPUs as it would for a single-level guest. After the guest boots up, the control of guest VCPUs can be transferred to/from an L1 using attach/detach operations.

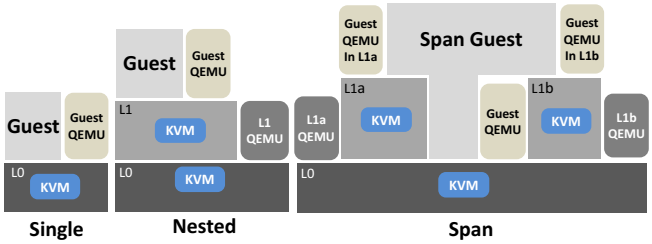


Figure 6: Roles of QEMU (Guest Controller) and KVM (hypervisor) for Single-level, Nested, and Span VMs.

7 Implementation Details

Platform and Modifications: Our prototype supports running an *unmodified* Linux guest as a Span VM in modes V_3, V_4, and V_5 from Figure 1. In our test setup, the guest runs Ubuntu 15.10 with Linux 4.2.0. The prototype for Span virtualization is implemented by modifying the KVM/QEMU nested virtualization support that is built into standard Linux distributions. Currently the implementation of L0 and all L1s uses modified KVM/QEMU hypervisors in Linux, specifically QEMU-1.2.0, kvm-kmod-3.14.2 and Linux 3.14.2. The modifications are different for the L0 and L1 layers. Ideally, we would prefer L1 to be unmodified to simplify its interface with L0. However, current hypervisors assume complete and exclusive guest control whereas Span allows L1s to exercise partial control over a subset of guest resources. Supporting partial guest control necessarily requires changes to L1 for attaching/detaching with a subset of guest resources and memory event subscription. In implementing L1 attach/detach operations on a guest, we tried, as much as possible, to reuse existing implementations of VM creation/termination operations.

Code size and memory footprint: Our implementation required about 2200 lines of code changes in KVM/QEMU, which is roughly 980+ lines in KVM and 500+ lines in QEMU for L0, 300+ in KVM and 200+ in QEMU for L1, and another 180+ in the virtio backend. We disabled unnecessary kernel components in both L0 and L1 implementations to reduce their footprint. When idle, L0 was observed to have 600MB usage at startup. When running an idle Span guest attached to an idle L1, L0's memory usage increased to 1756MB after excluding usage by the guest and the L1. The L1's initial memory usage, as measured from L0, was 1GB after excluding the guest footprint. This is an initial prototype to validate our ideas. The footprints of L0 and L1 implementations could be further reduced using one of many lightweight Linux distributions [14].

Guest Controller: A user-level control process, called the Guest Controller, runs on the hypervisor alongside each guest. In KVM/QEMU, the Guest Controller is a QEMU process which assists the KVM hypervisor with various control tasks on a guest, including guest initialization, I/O emulation, checkpointing, and migration. Figure 6 shows the position of the Guest Controller in different virtualization models. In both single-level and nested virtualization, there is only one Guest Controller per guest, since each guest is completely controlled by one hypervisor. Additionally, in the nested case, each L1 has its own Guest Controller that runs on L0. In Span virtualization, each guest is associated with multiple Guest Controllers, one per attached hypervisor. For instance, the Span Guest in Figure 6 is associated with three Guest Controllers, one each on L0, $L1_a$, and $L1_b$. During attach/detach operations, the Guest Controller in an L1 initiates the mapping/unmapping of guest memory into the L1's address space and, if needed, acquires/releases control over the guest's VCPU and virtual I/O devices.

Paravirtual I/O Architecture: The Guest Controller also performs I/O emulation of virtual I/O devices controlled by its corresponding hypervisor. The paravirtual device model described in Section 5 is called *virtio* in KVM/QEMU [54]. For nested guests, the virtio drivers are used at two levels: once between L0 and each L1 and again between an L1 and the guest. This design is also called *virtio-over-virtio*. A kick is implemented in virtio as a software trap from the frontend leading to a VM exit to KVM, which delivers the kick to the Guest Controller as a signal. Upon I/O completion, the Guest Controller requests KVM to inject a virtual interrupt into the guest. Kicks and interrupts are forwarded across hypervisors using the message channel. Redirected interrupts are received and injected into the guest by a modified version of KVM's virtual IOAPIC code.

VCPU Control: The Guest Controllers in different hypervisors communicate with the Guest Controller in L0 to acquire or relinquish guest VCPU control. The Guest Controller represents each guest VCPU as a user space thread. A newly attached L1 hypervisor does not initialize guest VCPU state from scratch. Rather, the Guest Controller in the L1 accepts a checkpointed guest VCPU state from its counterpart in L0 using a technique similar to that used for live VM migration between physical hosts. After guest VCPU states are transferred from L0 to L1, the L1 Guest Controller resumes the guest VCPU threads while the L0 Guest Controller pauses its VCPU threads. A VCPU detach operation similarly transfers a checkpoint of guest VCPU states from L1 to L0. Transfer of guest VCPU states from one L1 to another is presently accomplished through a combination of detaching the source L1 from the guest VCPUs followed by attaching to the destination L1 (although a direct transfer could be potentially more efficient).

Message Channel: The message channel between L0 and each L1 is implemented using a combination of hypercalls and UDP messages. Hypercalls from an L1 to L0 are used for attach/detach operations on guest

memory. UDP messages between an L1 and L0 are used for relaying I/O requests, device interrupts, memory subscription messages, and attach/detach operations on guest VCPU and I/O devices. UDP messages are presently used for ease of implementation and will be replaced by better alternatives such as hypercalls, callbacks, or shared buffers.

8 Evaluation

We first demonstrate unmodified Span VMs that can simultaneously use services from multiple L1s. Next we investigate how Span guests perform compared to traditional single-level and nested guests. Our setup consists of a server containing dual six-core Intel Xeon 2.10 GHz CPUs, 128GB memory and 1Gbps Ethernet. The software configurations for L0, L1s, and Span guests are as described earlier in Section 7. Each data point is a mean (average) over at least five or more runs.

8.1 Span VM Usage Examples

We present three examples in which a Span VM transparently utilizes services from multiple L1s. An unmodified guest is controlled by three coresident hypervisors, namely, L0, L1a, and L1b.

Use Case 1 – Network Monitoring and VM Introspection: In the first use case, the two L1s passively examine the guest state, while L0 supervises resource control. L1a controls the guest's virtual network device whereas L1b controls the guest VCPUs. L1a performs network traffic monitoring by running the `tcpdump` tool to capture packets on the guest's virtual network interface. Here we use `tcpdump` as a stand-in for other more complex packet filtering and analysis tools.

L1b performs VM introspection (VMI) using a tool called Volatility [3] which continuously inspects a guest's memory using a utility such as `pmemsave` to extract an accurate list of all processes running inside the guest. The guest OS is infected by a rootkit, Kernel Beast [38], which can hide malicious activity and present an inaccurate process list to the compromised guest. Volatility, running in L1b, can nevertheless extract an accurate guest process list using VM introspection.

Figure 7 shows a screenshot, where the top window shows the `tcpdump` output in L1a, specifically the SSH traffic from the guest. The bottom right window shows that the rootkit KBeast in the guest OS hides a process `evil`, i.e. it prevents the process `evil` from being listed using the `ps` command in the guest. The bottom left window shows that Volatility, running in L1b, successfully detects the process `evil` hidden by the KBeast rootkit in the guest.

This use case highlights several salient features of our design. First, an unmodified guest executes correctly

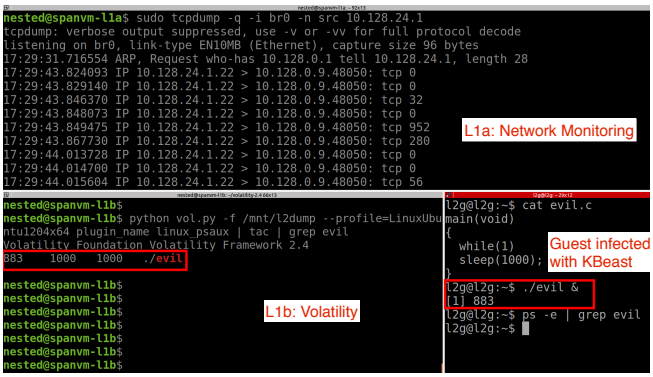
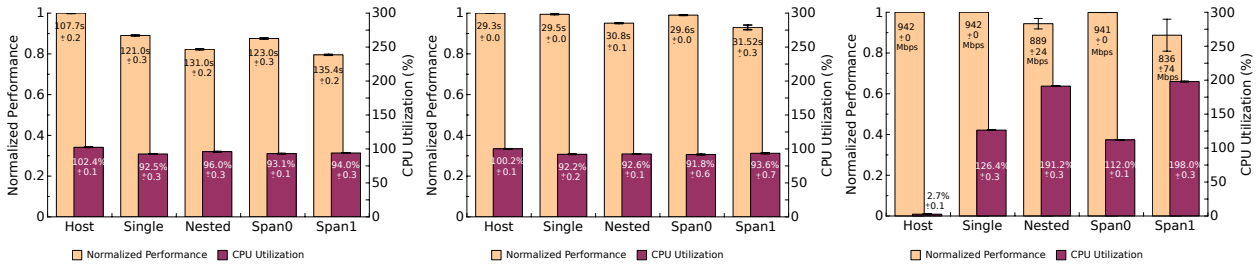


Figure 7: A screenshot of Span VM simultaneously using services from two L1s.

even though its resources are controlled by multiple hypervisors. Second, an L1 can transparently examine guest memory. Third, an L1 controlling a guest virtual device (here network interface) can examine all I/O requests specific to the device even if the I/O requests are initiated from guest VCPUs controlled by another hypervisor. Thus an I/O device can be delegated to an L1 that does not control the guest VCPUs.

Use Case 2 – Guest Mirroring and VM Introspection: In this use case, we demonstrate an L1 that subscribes to guest memory events from L0. Hypervisors can provide a high availability service that protects unmodified guests from a failure of the physical machine. Solutions, such as Remus [24], typically work by continually transferring live incremental checkpoints of the guest state to a remote backup server, an operation that we call *guest mirroring*. When the primary VM fails, its backup image is activated, and the VM continues running as if failure never happened. To checkpoint incrementally, hypervisors typically use a feature called dirty page tracking. The hypervisor maintains a dirty bitmap, i.e. the set of pages that were dirtied since the last checkpoint. The dirty bitmap is constructed by marking all guest pages read-only in the EPT and recording dirtied pages upon write traps. The pages listed in the dirty bitmap are incrementally copied to the backup server.

As a first approximation of guest mirroring, we modified the pre-copy live migration code in KVM/QEMU to periodically copy all dirtied guest pages to a backup server at a given frequency. In our setup, L1a mirrors a Span guest while L1b runs Volatility and controls guest VCPUs. L1a uses memory event subscription to track write events, construct the dirty bitmap, and periodically transfer any dirty pages to the backup server. We measured the average bandwidth reported by the iPerf [1] client benchmark running in the guest when L1a mirrors the guest memory at different frequencies. When guest mirroring happens every 12 seconds, iPerf delivers 800Mbps average bandwidth which is about the same as



(a) Kernbench (b) Quicksort (c) iPerf

Figure 8: No-op Mode: Normalized performance when no services run in host, L0, or L1s. The L0 controls the virtio block and network devices of the guest.

with a nested guest. When guest mirroring happens every second, the average bandwidth drops to 600Mbps, indicating a 25% performance impact of event subscription at very high mirroring frequencies.

Use Case 3 – Proactive Refresh: Hypervisor-level services may contain latent bugs, such as memory leaks, or other vulnerabilities that become worse over time, making a monolithic hypervisor unreliable for guests. Techniques like Microreboot[18] and ReHype[43] have been proposed to improve hypervisor availability, either proactively or post-failure. We have already seen how Span virtualization can compartmentalize unreliable hypervisor-level services in an isolated deprivileged L1. Here, we go one step further and proactively replace unreliable L1s with a fresh reliable instance while the guest and the base L0 hypervisor keep running. In our setup, an old L1 (L1a) was attached to a 3GB Span guest. To perform hypervisor refresh, we attached a new pre-booted replacement hypervisor (L1b) to the guest memory. Then L1a was detached from the guest by transferring guest VCPU and I/O devices to L1b via L0. In our implementation, the entire refresh operation from attaching L1b to detaching L1a completes on the average within 740ms. Of this, 670ms are spent in attaching L1b to guest memory while the guest is running. The remaining 70ms is the guest downtime due to the transfer of VCPU and I/O states. Thus Span virtualization achieves sub-second L1 refresh latency. If we attach the replacement L1b to guest memory well in advance, then the VCPU and I/O state transfer can be triggered on-demand by events, such as unusual memory pressure or CPU usage, yielding sub-100ms guest downtime and event response latency. In contrast, using pre-copy [22] to live migrate a guest from L1a to L1b can take several seconds depending on guest size and workload [65].

8.2 Macro Benchmarks

Here we compare the performance of macro benchmarks in Span VM against a native host (no hypervisor), single-level, and nested guests. Table 2 shows the memory and processor assignments at each layer for each case. The guest always has 3GB memory and one VCPU. L0 always has 128GB and 12 physical CPU cores. In the nested configuration, L1 has 16GB memory and 8 VCPUs. The guest VCPU in the Span0 configuration is controlled by L0, and in Span1 by an L1. Finally, in both Span0 and Span1, L1a and L1b each have 8GB of memory and 4VCPUs, so their sums match the L1 in the nested setting.

	L0		L1		L2	
	Mem	CPUs	Mem	VCPUs	Mem	VCPUs
Host	128GB	12	N/A	N/A	N/A	N/A
Single	128GB	12	3GB	1	N/A	N/A
Nested	128GB	12	16GB	8	3GB	1
Span0	128GB	12	8GB	4	3GB	1 on L0
Span1	128GB	12	8GB	4	3GB	1 on L1

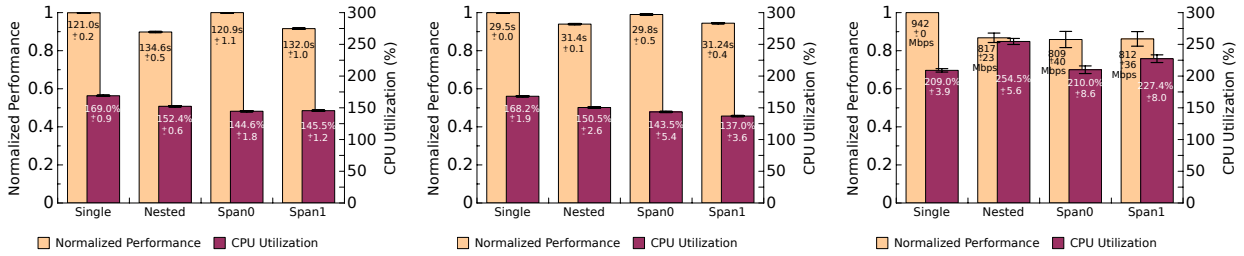
Table 2: Memory and CPU assignments for experiments.

The guest runs one of the following three benchmarks: (a) **Kernbench** [41] compiles the Linux kernel. (b) **Quicksort** sorts 400MB of data in memory. (c) **iPerf** [1] measures network bandwidth to another host.

The benchmarks run in two modes: *No-op Mode*, when no hypervisor-level services run, and *Service Mode*, when network monitoring and VM introspection services run at either L0 or L1s. The figures report each benchmark's normalized performance against the best case and system-wide average CPU utilization, which is measured in L0 using the `atop` command each second during experiments.

From Figures 8(a) and (b) and Figures 9(a) and (b), in both modes for Kernbench and Quicksort, Span0 performs comparably with the single-level setting and Span1 performs comparably with the nested setting, with similar CPU utilization.

For iPerf in No-op mode (Figure 8(c)), we observe that the Span1 guest experiences about 6% degradation over the nested guest with notable bandwidth fluctuation and 7% more CPU utilization. This is because the guest's VCPU in Span1 is controlled by L1a, but the guest's network device is controlled by L0. Hence, guest I/O requests (kicks) and responses are forwarded from L1a to L0 via the message channel. The message channel is currently implemented using UDP messages, which compete with guest's iPerf client traffic on the L1's vir-



(a) Kernbench (b) Quicksort (c) iPerf

Figure 9: Service Mode: Normalized performance with hypervisor-level services network monitoring and Volatility. For single-level, L0 runs both services. For nested, L1 runs both services. For Span0 and Span1, L1a runs network monitoring and controls the guest's network device; L1b runs Volatility; L0 controls guest's block device.

tio network interface with L0. We observed that if L1a controls the guest network device as well, then iPerf in the Span1 guest performs as well as in the nested guest.

For iPerf in service mode (Figure 9(c)), nested, Span0, and Span1 guests perform about 14–15% worse than the single-level guest, due to the combined effect of virtio-over-virtio overhead and `tcpdump` running in L1a. Further, for Span0, the guest VCPU is controlled by L0 whereas the network device is controlled by L1a. Thus forwarding of I/O kicks and interrupts between L0 and L1a via the UDP-based message channel balances out any gains from having guest VCPUs run on L0.

Figure 8(c) shows that the average CPU utilization increases significantly for iPerf in no-op mode – from 2.7% for the native host to 100+% for the single-level and Span0 configurations and 180+% for the nested and Span1 configurations. The increase appears to be due to the virtio network device implementation in QEMU, since we observed this higher CPU utilization even with newer versions of (unmodified) QEMU (v2.7) and Linux (v4.4.2). Figures 8(c) and 9(c) also show higher CPU utilization for the nested and Span1 cases compared to the single-level case. This is because guest VCPUs are controlled by L1s in the nested and Span1 cases, making nested VM exits more expensive.

8.3 Micro Benchmarks

Attach Operation: Figure 10 shows the time taken to attach an L1 to a guest's memory, VCPU, and I/O devices as the guest memory size is increased. The time taken to attach memory of a 1GB Span guest is about 220ms. Memory attach overhead increases with guest size because each page that L1 has allocated for Span needs to be remapped to the Span physical page in L0.

Attaching VCPUs to one of the L1s takes about 50ms. Attaching virtual I/O devices takes 135ms. When I/O control has to be transferred between hypervisors, the VCPUs need to be paused. The VCPUs could be running on any of the L1s and hence L0 needs to coordinate pausing and resuming the VCPUs during the transfer. The

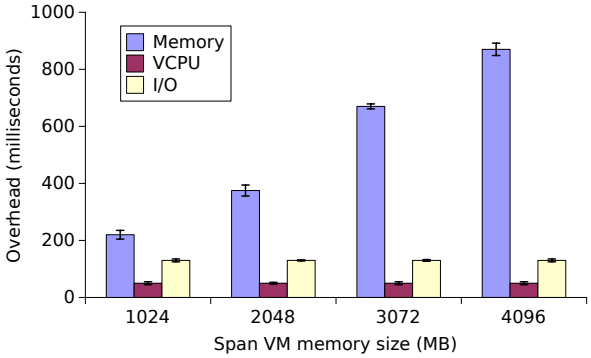


Figure 10: Overhead of attaching an L1 to a guest.

	Single	Nested	Span
EPT Fault	2.4	2.8	3.3
Virtual EPT Fault	-	23.3	24.1
Shadow EPT Fault	-	3.7	4.1
Message Channel	-	-	53
Memory Event Notify	-	-	103.5

Table 3: Low-level latencies(μs) in Span virtualization.

detach operation for VCPUs and I/O devices has similar overhead.

Page Fault Servicing: Table 3 shows the latency of page fault handling and message channel. We measured the average service times for EPT faults in Span at both levels of nesting. It takes on the average 3.3μs to resolve a fault caused against EPT_{L1} and on the average 24.1μs to resolve a fault against the Virtual EPT. In contrast, the corresponding values measured for the nested case are 2.8μs and 23.3μs. For the single-level case, EPT-fault processing takes 2.4μs. The difference is due to the extra synchronization work in the EPT-fault handler in L0.

Message Channel and Memory Events: The message channel is used in Span virtualization to exchange events and requests between L0 and L1s. It takes on the average 53μs to send a message between L0 and an L1. We also measured the overhead of notifying L1 subscribers from L0 for write events on a guest page. Without any subscribers, the write-fault processing takes on the average 3.5μs in L0. Notifying the write event over

the message channel from L0 to an L1 subscriber adds around 100μs, including a response from L1.

9 Related Work

Here, we review prior work on user space services, service VMs, and nested virtualization. We build upon earlier discussion of their relative merits in Section 2.

User space Services: Microkernels and library operating systems have a long history [44, 13, 28, 35] of providing OS services in user space. μDenali [63] allows programmers to use event interposition to extend the hypervisor with new user-level services such as disk and network I/O. In the KVM/QEMU [40] platform, each guest is associated with a dedicated user space management process, namely QEMU. A single QEMU process bundles multiple services for its guest such as VM launch/exit/pause, paravirtual I/O, migration, and checkpointing. One can associate different variants of QEMU with different guests, allowing some degree of service customization. However, QEMU's interface with the KVM hypervisor is large, consisting of system calls, signals, and shared buffers with the kernel, which increases the KVM hypervisor's exposure to potentially untrusted services. Also, while user space services can map guest memory and control paravirtual I/O, they lack direct control over low-level guest resources such as EPT mappings and VCPU scheduling, unlike nesting and Span.

Service VMs: Another option is to provide guest services via specialized Service VMs that run alongside the guest. For instance, the Xen [4] platform runs a trusted service VM called Dom0 which runs paravirtualized Linux, controls all guests via hypercalls to the Xen hypervisor, and provides guests with services related to lifecycle management and I/O. To avoid a single point of failure or vulnerability, the Xoar [47, 23] project proposed decomposing Dom0 into smaller service domains, one per service, that can be replaced or restarted. Possible support for third-party service domains has been discussed [16], but its status is unclear. Nova [58] minimizes the size of the hypervisor by implementing the VMM, device drivers, and special-purpose applications in user space. Self-service clouds [17] allows users to customize control over services used by their VMs on untrusted clouds. Services, such as storage and security, can be customized by privileged service domains, whereas the hypervisor controls all low-level guest resources, such as VCPUs and EPT mappings.

Nested Virtualization: Nested virtualization was originally proposed and refined in the 1970s [32, 33, 51, 8, 9, 48] and has experienced renewed interest in recent years [29, 34, 10]. Recent support [25], such as VMCS shadowing [62] and direct device assignment [67] aim to reduce nesting overheads related to VM exits and I/O.

Nesting enables vertical stacking of two layers of hypervisor-level services. Third parties such as Ravello [2] and XenBlanket [66, 57] leverage nesting to offer hypervisor-level services (in an L1) over public cloud platforms (L0) such as EC2 and Azure, often pitching their service as a way to avoid lock-in with a cloud provider. However, this model also leads to a different level of lock-in, where a guest is unable use services from more than one third party. Further, these third-party services are not fully trusted by the base hypervisor (L0) of the cloud provider, necessitating the use of nesting, rather than user space services. Span virtualization prevents guest lock-in at all levels by adding support for multiple third-party L1s to concurrently service a guest, while maintaining the isolation afforded by nesting.

Ephemeral virtualization [65] combines nesting and optimized live migration [22, 36] to enable transient control over guest by L1s. L1s and L0 take turns exchanging full control over the guest by co-mapping its memory. In contrast, Span allows multiple L1s to concurrently exercise either full or partial control over a guest, in either continuous or transient modes.

10 Conclusions

A rich set of hypervisor-level services have been proposed in recent years, such as VM introspection, high availability, live patching, and migration. However, in modern cloud platforms, a sole controlling hypervisor continues to host all such services. Specifically, support for third-parties to offer hypervisor-level services to guests is lacking. We presented a new approach, called *Span virtualization*, which leverages nesting to enable multiple coresident, but isolated, hypervisors to control and service a common guest. Our prototype of Span virtualization on the KVM/QEMU platform can support unmodified guests which simultaneously use multiple services that augment the base hypervisor's functionality. Span guests achieve performance comparable to traditional nested guests. Looking ahead, we believe that the opportunity for a cloud-based ecosystem of hypervisor-level services is large, including security services, cross-cloud portability, custom schedulers, virtual devices, and high availability.

11 Acknowledgement

This work was funded in part by the National Science Foundation via awards 1527338 and 1320689. We thank our shepherd, Nadav Amit, and all reviewers for insightful feedback; Umesh Deshpande, Spoorti Doddamani, Michael Hines, Hani Jamjoom, Siddhesh Phadke, and Piush Sinha, for discussions, implementation, and evaluation; and the Turtles Project [10] authors for inspiration.

References

[1] iPerf: The Network Bandwidth Measurement Tool. http://iperf.fr/.

[2] Ravello Systems. https://www.ravellosystems.com/.

[3] Volatility Framework. http://www.volatilityfoundation.org/.

[4] Xen Hypervisor. http://www.xen.org/.

[5] AMD. AMD Virtualization (AMD-V). http://www.amd.com/en-us/solutions/servers/virtualization.

[6] 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 *Proc. of SOSP* (Bolton Landing, NY, USA, 2003), pp. 164–177.

[7] BEHAM, M., VLAD, M., AND REISER, H. Intrusion detection and honeypots in nested virtualization environments. In *Proc. of 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks(DSN)* (Budapest, Hungary, June 2013).

[8] BELPAIRE, G., AND HSU, N.-T. Formal properties of recursive virtual machine architectures. In *Proc. of SOSP, Austin, Texas, USA* (1975), pp. 89–96.

[9] BELPAIRE, G., AND HSU, N.-T. Hardware architecture for recursive virtual machines. In *Proc. of Annual ACM Conference* (1975), pp. 14–18.

[10] BEN-YEHUDA, M., DAY, M. D., DUBITZKY, Z., FACTOR, M., HAR'EL, N., GORDON, A., LIGUORI, A., WASSERMAN, O., AND YASSOUR, B.-A. The Turtles project: Design and implementation of nested virtualization. In *Proc. of Operating Systems Design and Implementation* (2010).

[11] BEN-YEHUDA, M., MASON, J., XENIDIS, J., KRIEGER, O., VAN DOORN, L., NAKAJIMA, J., MALLICK, A., AND WAHLIG, E. Utilizing IOMMUs for virtualization in Linux and Xen. In *Ottawa Linux Symposium* (July 2006).

[12] BEN-YEHUDA, M., XENIDIS, J., OSTROWSKI, M., RISTER, K., BRUEMMER, A., AND VAN DOORN, L. The price of safety: Evaluating IOMMU performance. In *Proc. of Ottawa Linux Symposium* (July 2007).

[13] BERSHAD, B. N., CHAMBERS, C., EGGERS, S., MAEDA, C., MCNAMEE, D., PARDYAK, P., SAVAGE, S., AND SIRER, E. G. SPIN : An extensible microkernel for application-specific operating system services. *Proc. of ACM SIGOPS Operating Systems Review 29*, 1 (1995), 74–77.

[14] BHARTIYA, S. Best lightweight linux distros for 2017. https://www.linux.com/news/best-lightweight-linux-distros-2017.

[15] BROMIUM. https://www.bromium.com.

[16] BULPIN, J. Whatever happened to XenServer's Windsor architecture? https://xenserver.org/blog/entry/whatever-happened-to-xenserver-s-windsor-architecture.html.

[17] BUTT, S., LAGAR-CAVILLA, H. A., SRIVASTAVA, A., AND GANAPATHY, V. Self-service cloud computing. In *Proc. of ACM Conference on Computer and Communications Security(CCS)* (Raleigh, NC, USA, 2012).

[18] CANDEA, G., KAWAMOTO, S., FUJIKI, Y., FRIEDMAN, G., AND FOX, A. Microreboot-a technique for cheap recovery. In *Proc. of 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (San Francisco, CA, USA, 2004), vol. 4, pp. 31–44.

[19] CHEN, H., CHEN, R., ZHANG, F., ZANG, B., AND YEW, P. Live updating operating systems using virtualization. In *Proc. of ACM International Conference on Virtual Execution Environments (VEE)* (Ottawa, Canada, June 2006).

[20] CITRIX. *XenDesktop*. https://www.citrix.com/products/xenapp-xendesktop/.

[21] CITRIX. *XenDesktop and The Evolution of Hardware-Assisted Server Technologies*. https://www.citrix.com/content/dam/citrix/en_us/documents/go/2015-edition-hosted-desktop.pdf.

[22] CLARK, C., FRASER, K., HAND, S., HANSEN, J., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *Proc. of Network System Design and Implementation* (2005).

[23] COLP, P., NANAVATI, M., ZHU, J., AIELLO, W., COKER, G., DEEGAN, T., LOSCOCCO, P., AND WARFIELD, A. Breaking up is hard to do: Security and functionality in a commodity hypervisor. In *Proc. of SOSP* (2011), pp. 189–202.

[24] CULLY, B., LEFEBVRE, G., MEYER, D., FEELEY, M., HUTCHINSON, N., AND WARFIELD, A. Remus: High availability via asynchronous virtual machine replication. In *Proc. of Networked Systems Design and Implementation, San Francisco, CA, USA* (2008).

[25] DAS, B., ZHANG, Y. Y., AND KISZKA, J. Nested virtualization: State of the art and future directions. In *KVM Forum* (2014).

[26] DINABURG, A., ROYAL, P., SHARIF, M., AND LEE, W. Ether: malware analysis via hardware virtualization extensions. In *Proc. of 15th ACM conference on Computer and communications security (CCS)* (Alexandria, VA, USA, 2008), pp. 51–62.

[27] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proc. of 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Boston, MA, Dec. 2002).

[28] ENGLER, D. R., KAASHOEK, M. F., ET AL. Exokernel: An operating system architecture for application-level resource management. In *ACM SIGOPS Operating Systems Review* (1995), vol. 29(5), pp. 251–266.

[29] FORD, B., HIBLER, M., LEPREAU, J., TULLMANN, P., BACK, G., AND CLAWSON, S. Microkernels Meet Recursive Virtual Machines. In *Proc. OSDI, Seattle, Washington, USA* (1996), pp. 137–151.

[30] GARFINKEL, T., AND ROSENBLUM, M. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Network & Distributed Systems Security Symposium* (San Diego, CA USA, 2003).

[31] GEBHARDT, C., AND DALTON, C. LaLa: A Late Launch Application. In *Workshop on Scalable Trusted Computing, Chicago, Illinois, USA* (2009), pp. 1–8.

[32] GOLDBERG, R. P. Architecture of Virtual Machines. In *Proceedings of the Workshop on Virtual Computer Systems, Cambridge, MA, USA* (1973), pp. 74–112.

[33] GOLDBERG, R. P. Survey of Virtual Machine Research. *Computer 7*, 6 (1974), 34–45.

[34] GRAF, A., AND ROEDEL, J. Nesting the virtualized world. In *Linux Plumbers Conference* (Sept. 2009).

[35] HAND, S., WARFIELD, A., FRASER, K., KOTSOVINOS, E., AND MAGENHEIMER, D. J. Are virtual machine monitors microkernels done right? In *Proc. of HotOS* (2005).

[36] HINES, M., DESHPANDE, U., AND GOPALAN, K. Post-copy live migration of virtual machines. *In SIGOPS Operating Systems Review* (July 2009), 14–26.

[37] INTEL CORP. *Intel 64 and IA-32 Architecture Software Developers Manual, Volume 3, System Programming Guide. Order number 325384.* April 2016.

[38] IPSECS. Kernel Beast. `http://core.ipsecs.com/rootkit/kernel-rootkit/`.

[39] JIANG, X., WANG, X., AND XU, D. Stealthy malware detection and monitoring through VMM-based "out-of-the-box" semantic view reconstruction. *ACM Trans. Information Systems Security 13*, 2 (Mar. 2010), 1–28.

[40] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. KVM: the linux virtual machine monitor. In *Proc. of Linux Symposium* (June 2007).

[41] KOLIVAS, C. Kernbench. `http://ck.kolivas.org/apps/kernbench/`.

[42] KOURAI, K., AND CHIBA, S. Hyperspector: Virtual distributed monitoring environments for secure intrusion detection. In *ACM/USENIX International Conference on Virtual Execution Environments* (2005), pp. 197 – 207.

[43] LE, M., AND TAMIR, Y. ReHype: enabling VM survival across hypervisor failures. In *Proc. of ACM SIGPLAN Notices* (2011), vol. 46, ACM, pp. 63–74.

[44] LIEDTKE, J. On micro-kernel construction. *Proc. of ACM SIGOPS Operating Systems Review 29*, 5 (1995), 237–250.

[45] LOWELL, D. E., SAITO, Y., AND SAMBERG, E. J. Devirtualizable virtual machines enabling general, single-node, online maintenance. *Proc. of SIGARCH Comput. Archit. News 32*, 5 (Oct. 2004), 211–223.

[46] MCAFEE. Root Out Rootkits: An Inside Look at McAfee Deep Defender. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/mcafee-deep-defender-deepsafe-rootkit-protection-paper.pdf.

[47] MURRAY, D. G., MILOS, G., AND HAND, S. Improving Xen Security Through Disaggregation. In *Proc. of Virtual Execution Environments* (2008), pp. 151–160.

[48] OSISEK, D. L., JACKSON, K. M., AND GUM, P. H. Esa/390 interpretive-execution architecture, foundation for vm/esa. *IBM Systems Journal 30*, 1 (Feb. 1991), 34–51.

[49] PAYNE, B. D., CARBONE, M., SHARIF, M., AND LEE, W. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *Proc. of IEEE Symposium on Security and Privacy* (2008), pp. 233 – 247.

[50] PCI SIG. Single Root I/O Virtualization and Sharing. http://www.pcisig.com/specifications/iov/single_root/.

[51] POPEK, G. J., AND GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. *Proc. of Communications of ACM 17*, 7 (July 1974), 412–421.

[52] REAL TIME SYSTEMS GMBH. *RTS Real-Time Hypervisor*. http://www.real-time-systems.com/products/index.php.

[53] RILEY, R., JIANG, X., AND XU, D. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In *Proc. of Recent Advances in Intrusion Detection* (2008), pp. 1–20.

[54] RUSSELL, R. Virtio: Towards a de-facto standard for virtual I/O devices. *Proc. of SIGOPS Operating Systems Review 42*, 5 (July 2008), 95–103.

[55] RUTKOWSKA, J. Subverting vista kernel for fun and profit. In *Blackhat* (Las Vegas, USA, Aug. 2006).

[56] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proc. of ACM SIGOPS Operating Systems Review* (2007), vol. 41(6), pp. 335–350.

[57] SHEN, Z., JIA, Q., SELA, G.-E., RAINERO, B., SONG, W., VAN RENESSE, R., AND WEATHERSPOON, H. Follow the Sun Through the Clouds: Application Migration for Geographically Shifting Workloads. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (2016), pp. 141–154.

[58] STEINBERG, U., AND KAUER, B. NOVA: A Microhypervisor-based Secure Virtualization Architecture. In *Proc. of EuroSys*, pp. 209–222.

[59] SUGERMAN, J., VENKITACHALAM, G., AND LIM, B.-H. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proc. of USENIX Annual Technical Conference, Monterey, CA, USA* (2002).

[60] SUNEJA, S., ISCI, C., BALA, V., DE LARA, E., AND MUMMERT, T. Non-intrusive, Out-of-band and Out-of-the-box Systems Monitoring in the Cloud. In *Proc. of SIGMETRICS'14, Austin, TX, USA* (2014).

[61] TOLDINAS, J., RUDZIKA, D., ŠTUIKYS, V., AND ZIBERKAS, G. Rootkit Detection Experiment within a Virtual Environment. *Electronics and Electrical Engineering–Kaunas: Technologija*, 8 (2009), 104.

[62] WASSERMAN, O. Nested Virtualization: Shadow Turtles. In *KVM Forum, Edinburgh, Spain* (October 2013).

[63] WHITAKER, A., COX, R., AND SHAW, M. Constructing services with interposable virtual hardware. In *Proc. of First USENIX Symposium on Networked Systems Design and Implementation* (San Francisco, California, 2004).

[64] WIKIPEDIA. Phoenix Hyperspace. https://en.wikipedia.org/wiki/HyperSpace_(software).

[65] WILLIAMS, D., HU, Y., DESHPANDE, U., SINHA, P. K., BILA, N., GOPALAN, K., AND JAMJOOM, H. Enabling efficient hypervisor-as-a-service clouds with ephemeral virtualization. In *Proc. of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)* (2016).

[66] WILLIAMS, D., JAMJOOM, H., AND WEATHERSPOON, H. The Xen-Blanket: Virtualize once, run everywhere. In *EuroSys, Bern, Switzerland* (2012).

[67] YASSOUR, B.-A., BEN-YEHUDA, M., AND WASSERMAN, O. Direct Device Assignment for Untrusted Fully-Virtualized Virtual Machines. Tech. rep., IBM Research, 2008.

[68] ZHANG, F., CHEN, J., CHEN, H., AND ZANG, B. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proc. of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 203–216.

Preemptive, Low Latency Datacenter Scheduling via Lightweight Virtualization

Wei Chen[1], Jia Rao[2], and Xiaobo Zhou[1]

[1]University of Colorado, Colorado Springs, {cwei, xzhou}@uccs.edu
[2]University of Texas at Arlington, jia.rao@uta.edu

Abstract

Data centers are evolving to host heterogeneous workloads on shared clusters to reduce the operational cost and achieve higher resource utilization. However, it is challenging to schedule heterogeneous workloads with diverse resource requirements and QoS constraints. On the one hand, latency-critical jobs need to be scheduled as soon as they are submitted to avoid any queuing delays. On the other hand, best-effort long jobs should be allowed to occupy the cluster when there are idle resources to improve cluster utilization. The challenge lies in how to minimize the queuing delays of short jobs while maximizing cluster utilization. Existing solutions either forcibly kill long jobs to guarantee low latency for short jobs or disable preemption to optimize utilization. Hybrid approaches with resource reservations have been proposed but need to be tuned for specific workloads.

In this paper, we propose and develop BIG-C, a container-based resource management framework for Big Data cluster computing. The key design is to leverage lightweight virtualization, a.k.a, containers to make tasks preemptable in cluster scheduling. We devise two types of preemption strategies: *immediate* and *graceful* preemptions and show their effectiveness and tradeoffs with loosely-coupled MapReduce workloads as well as iterative, in-memory Spark workloads. Based on the mechanisms for task preemption, we further develop a preemptive fair share cluster scheduler. We have implemented BIG-C in YARN. Our evaluation with synthetic and production workloads shows that low-latency and high utilization can be both attained when scheduling heterogeneous workloads on a contended cluster.

1 Introduction

Recently, the proliferation of data-intensive cluster applications, such as data mining, data analytics, scientific computation, and web search has led to the development of datacenter-scale computing. Resource efficiency is a critical issue when operating such datacenters at scale. Studies [5, 20, 21, 35] have shown that increasing utilization by sharing the hardware infrastructure among multiple users leads to superior resource and energy efficiencies. Therefore, cluster management frameworks, such as [16, 29, 31, 33] face the challenges of efficiently hosting a variety of heterogeneous workloads with diverse QoS requirements and resource demands.

Short jobs have stringent latency requirements and are sensitive to scheduling delays while long jobs can tolerate long latency but have higher requirements for the quality of scheduling, e.g., preserving data locality. To reconcile the conflicting objectives, recent proposed schedulers [4, 10] reserve a portion of the cluster to run exclusively short jobs using distributed scheduling while long jobs are scheduled onto the unreserved portion using centralized scheduling. The challenge is to determine the optimal partition of the cluster to guarantee low latency to short jobs while maintaining high cluster utilization, under highly dynamic workloads.

We look at the cluster scheduling problem from a different angle – if tasks from short jobs can preempt any long tasks, their scheduling can be made simple and fast while long jobs can run on any server in the cluster to maintain high utilization. Unfortunately, existing cluster schedulers do not support efficient task preemption. For example, YARN [31] and Mesos [16] only support kill-based preemption and killed tasks need to be restarted. This could lead to substantial slowdown to long-running jobs due to the loss of execution progress.

In this paper, we leverage application containers, a form of lightweight virtualization, to enable preemptive and low latency scheduling in clusters with heterogeneous workloads. Although containers, such as Docker, are being increasingly adopted in distributed systems [6, 33, 34], their usage is primarily for agile application deployment, leaving much of containers' potential in resource management unexploited. We explore the flexible resource management provided by container virtualization to enable low-cost task preemp-

tion. Specifically, tasks encapsulated in containers can be suspended by depriving the resources allocated to their containers and resumed later by replenishing the resources. Based on this preemption mechanism, we propose and develop BIG-C, a Container-based resource management framework for BIG data analytics that provides low-latency scheduling to preempting jobs while minimizing performance penalty to preempted jobs.

BIG-C uses Docker [22] to containerize tasks and relies on Linux `cgroups` to precisely control the CPU and memory allocations to such containers. In BIG-C, long jobs are immediately preempted upon the arrival of short jobs to guarantee low latency. We devise two types of preemptions: *immediate* and *graceful* preemptions. Immediate preemption instantaneously reduces the resources of the preempted task to a minimum footprint while still keeping the task alive to the task management. Graceful preemption gradually takes resources away from long tasks, minimizing long job slowdown. The two container-based preemption schemes replace the kill-based task preemption and can be seamlessly integrated into any fair sharing cluster schedulers and are transparent to applications. In BIG-C, we integrate the two preemption schemes into a preemptive fair share scheduler based on YARN's capacity scheduler.

We have implemented BIG-C on Apache YARN and evaluated it on a 26-node cluster using heterogeneous workloads composed of TPC-H queries and batch jobs from HiBench [17]. Experimental results show that BIG-C strikes a balance between short job latency and cluster utilization compared with state-of-the-art schedulers. The source code of BIG-C is publicly available [1].

Our results also provide insights on serving heterogeneous workloads in MapReduce and Spark. Immediate preemption works generally well for MapReduce jobs: (1) tasks are loosely coupled and the preemption of some tasks does not impede the progress of other tasks; (2) tasks have dedicated containers and their intermediate results are periodically persisted to disk, making it faster to reclaim their memory when preempted. In contrast, immediate preemption incurs significant performance loss to jobs in Spark: (1) Spark executors employ a multi-threaded model and the preemption of one container affects multiple tasks; (2) for Spark jobs with iterative computation, tasks involve in frequent synchronization between each other within and across executors/-containers; (3) due to in-memory processing, reclaiming memory from preempted containers may incur significant memory swapping. We find that graceful preemption is more suitable for Spark workloads.

[1] `https://github.com/yncxcw/big-c`

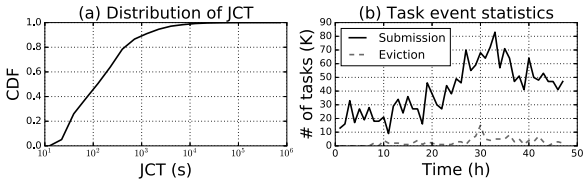


Figure 1: The analysis of Google's trace. (a) Job completion time (JCT). (b) Job submission and eviction rates.

2 Motivation

2.1 Real-world Trace Analysis

To understand job characteristics and resource usage of production workloads, we analyze the publicly available traces [28, 7] from Google data centers. Figure 1(a) shows that short jobs that complete within 1 minute dominate the traces and contribute to 80% of the total number of jobs. Although data center workloads mostly consist of short jobs, long jobs account for most resource usage. It has been reported by other researchers [10, 18, 27] that the top 10% long jobs, which are only responsible for 28% of the total number of tasks, account for more than 80% of the total task execution time. According to the analysis, there is a difficult tradeoff in data center scheduling: short jobs are sensitive to delays and critical to QoS enforcements while long jobs are important to maintaining high resource utilization. Existing research [11, 20, 21] shows that prioritizing short jobs and serving long jobs in a best-effort manner on a shared infrastructure meet both requirements.

When resources are contended, both prioritization and the enforcement of fair sharing can lead to the preemption of already running tasks. Preempted tasks are often evicted or killed. Figure 1(b) plots task submission and eviction rates in the Google trace for a period of 48 hours. According to Google, a task can be evicted by higher priority tasks in the case of resource shortage or stopped if the task owner user/group exceeds its fair share of cluster resources. Figure 1(b) shows that task eviction rate climbs up as submission rate increases, indicating a clear relationship between task preemptions and resource contention. In the 48-hour period, there were 910,099 tasks submitted, of which 93,714 were evicted and most of them were long jobs. As discussed above, long jobs account for a disproportionate amount of resource usage. The evictions of long jobs are especially expensive as the loss of their execution progress translates to significant resource waste. While evicted jobs can be relaunched when there are sufficient resources, the eviction can also add substantial delay to the completion time of such jobs.

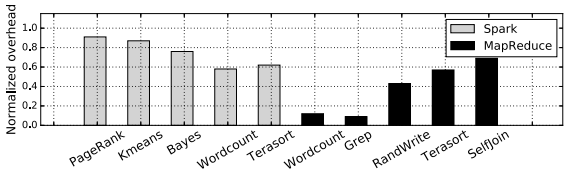


Figure 2: The overhead of kill-based task preemption in different types of workloads.

2.2 Overhead of Kill-based Preemption

Task killing is a simple means to realize preemption. However, killed tasks cannot be resumed and have to be relaunched. Most cluster schedulers use this approach due to its simplicity.

Figure 2 shows the overhead of kill-based preemption for different types of MapReduce and Spark jobs. We configured long jobs to fully utilize a 26-node YARN cluster. Detailed configuration of the cluster can be found in §5.1. During the execution of each long job, we injected a 6-minute burst of short Spark-SQL queries. The YARN default capacity scheduler was set to assign a share of 95% cluster resources to Spark-SQL queries, enforcing a strictly higher priority for the short jobs. Upon the arrival of short jobs, YARN kills tasks selected randomly from the long job to free resources needed by short jobs. Killed tasks are immediately resubmitted to YARN for rescheduling.

As shown in Figure 2, MapReduce jobs suffer less performance penalty from task killing than Spark jobs do. Task killing degraded the overall performance of MapReduce jobs by 8% - 64% while incurring as much as 92% overhead to Spark jobs. Among MapReduce jobs, those are dominated by the map phase, e.g., *wordcount*, suffered marginal degradation compared to the noticeable slowdown experienced by reduce-heavy jobs, e.g., *terasort*. Because mappers are usually small and independent from each other, the termination of a few mappers does not lead to much computation loss nor significantly delay job completion. In contrast, reducers require all-to-all communications with mappers. This data shuffling phase runs much longer than mappers. Therefore, the killing of one reducer requires the lengthy and resource intensive shuffling process to be restarted, which substantially delays job completion.

Spark jobs are more susceptible to delays due to task killings for the following reasons. First, Spark jobs, especially machine learning algorithms that iterate over a data set, require frequent synchronizations between tasks. If one task is killed, other dependent tasks are unable to make any progress. Second, Spark in-memory processing does not persist intermediate results to storage. For jobs with multiple stages, the killing of one task could lead to the re-computation of dependent stages. This recovery process is usually quite expensive. Figure 2 shows that Spark jobs suffered on average 70% slowdown when interrupted by the burst of short jobs.

3 Container-based Task Preemption

In this section, we present two simple container-based approaches for task preemption and in the next section we integrate them into cluster scheduling.

3.1 Container-based Virtualization

Container-based virtualization, such as Docker, has gained popularity due to its almost negligible overhead compared to hypervisor-based virtualization. A container provides isolated namespaces for applications running inside the container and forms a resource accounting and allocation unit. Linux uses control groups (`cgroups`) to precisely control the resource allocation to a container. Not only priorities can be set to reflect the relative importance of containers, hard resource limits guarantee that containers consume resources no more than a predefined upper bound even there are available resources in the system.

3.2 Immediate Task Preemption

We leverage the flexible resource allocation enabled by containers to temporarily suspend a task in Big Data analytics and reclaim its resources without losing the execution progress. We assume that each task is encapsulated into a container [2]. Each container forms a `cgroup` and is configured with two types of resources: CPU and memory. Parameter `cpuset.cpus` controls the number of CPU cores that a container can use and parameter `memory.limit_in_bytes` limits the maximum memory usage. `cpu.cfs_quota_us` and `cpu.cfs_period_us` together determine the maximum CPU allocation to a container. This enables fine-grained control of CPU cycles beyond allocating CPU cores.

Task suspension involves two steps: *stop task execution* and *save task context*. To stop a task, the host container is deprived of CPU to stop task execution. To save the task's context for later resumption, its dirty data in memory needs to be written back to disk. Fortunately, no additional effort is needed to support context saving. When reclaiming a container's memory, the virtual memory management in the host operating system (OS) writes back dirty data. For fault tolerance, cluster schedulers monitor the progress of individual tasks and launch speculative tasks if stragglers are detected.

The suspension of containers will falsely trigger the failover. To avoid extensive changes to cluster schedulers to support task preemption, we suspend a task but

[2] Spark runs multi-threaded task in an executor. Therefore, a container corresponds to an executor in Spark and contains multiple tasks.

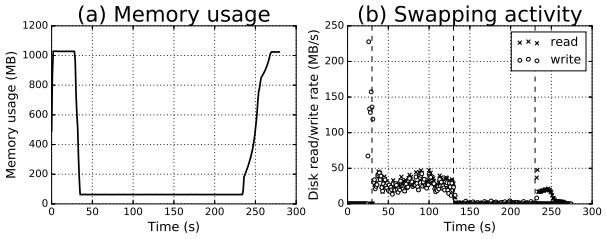


Figure 3: Immediate preemption incurs high overhead due to memory reclaiming and restoring.

maintain a minimal footprint for the task to keep it alive to the cluster scheduler. We empirically set the minimal container footprint to 1% CPU and 64 MB memory, with which the thread responsible for sending the heartbeat in the container still appears to be alive to the scheduler. We also disable speculative execution for suspended tasks.

Task resumption is simply re-activating the container by restoring its deprived resources. The resumption also follows two steps. The memory size of the container is restored from the minimal footprint back to its original size and the CPU limit is lifted. We call this type of preemption, which reclaims and restores all resources of a preempted task in one pass, *immediate preemption* (IP).

Overhead of immediate preemption Despite that kill-based preemption is crude, it guarantees timely scheduling of short jobs. The container-based immediate preemption, however, can possibly delay short job scheduling and inflict performance degradation to long jobs. First, it may take non-negligible time to reclaim the memory of preempted tasks before short tasks can be scheduled, depending on the working set size of preempted tasks. Second, task resumption requires loading saved context into memory. For certain jobs, this process is particularly long.

Figure 3 shows the memory swapping activities when a 1 GB container was suspended to the minimal footprint and later resumed. The container ran a multi-threaded synthetic Java benchmark that repeatedly and randomly touched a 1 GB array. Figure 3(a) shows that it took 3 seconds (between the 40th and 45th seconds) to reclaim nearly 1 GB memory. Note that the swapping activities lasted much longer until the container was deprived of CPU at the 130th second. It took even longer to load saved context into memory after the container was resumed at the 230th second. The reason is that the multiple threads in the container simultaneously loaded their working sets when memory was restored, resulting in a large volume of random disk access. The synthetic benchmark provides following insights on IP overhead:

- It is expensive to reclaim memory from a container that is actively dirtying its working set.
- Depriving CPU effectively throttles disk reads during memory reclaiming, shortening the suspension delay.
- Spark jobs are particularly susceptible to the resumption overhead when multiple tasks from an/a executor/container are activated to simultaneously load their working sets from disk.

To reduce the overhead, one optimization is to first reclaim CPU from a container to throttle task activity before memory is reclaimed. However, this optimization is not sufficient to guarantee short job latency or minimize long job degradation, which motivated us to develop the graceful preemption.

3.3 Graceful Task Preemption

While immediate preemption deprives a task of all resources to completely suspend the task, *graceful preemption* (GP) shrinks a preempted task and reclaims its resources in multiple rounds.

Compared to immediate preemption, graceful preemption reclaims a task's resources at a pre-defined step $\vec{r} = (c, m)$, where c and m are the unit resource reclamation for CPU cores and memory, respectively. GP is based on the following insights:

- Tasks from long jobs are usually larger than tasks from short jobs. Launching a short task often does not need to reclaim all resources of a long task.
- Resource slack is common in cluster computing. Memory slack could come from intentional overprovisioning at job launch to avoid Out-Of-Memory errors, dynamic and epochal memory demands at different job stages [24], or diminishing memory demands towards job completion. CPU slack is due to similar reasons. Even if CPU is fully utilized, gracefully reducing CPU allocation does not cause drastic performance degradation to a preempted task.
- Since tasks' requests are usually based on their peak demands, the partially reclaimed resources from preempted tasks are often sufficient for high priority tasks to make progress at an early stage of execution.

To reclaim resources from a preempted task, the cluster scheduler controls the iteration of graceful preemption. This process stops if the demands of the high priority tasks are satisfied. Similar to immediate preemption, in which we deprive the container of CPU to throttle task execution so as to accelerate memory reclamation, graceful preemption freezes a container's CPU when swapping activities are detected. Graceful preemption will continue until new tasks' demands are met.

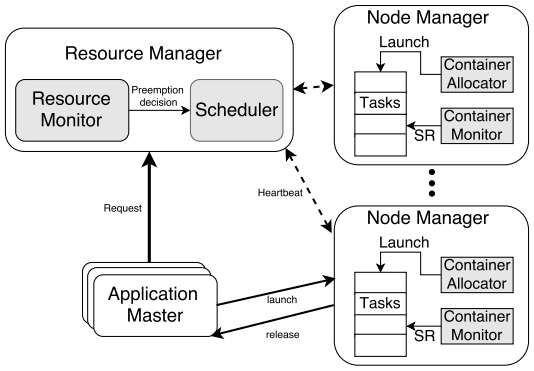


Figure 4: The architecture of BIG-C.

4 BIG-C: Preemptive Cluster Scheduling

In this section, we describe how to integrate container-based task preemption into YARN. We begin with a brief overview of YARN's resource management and task scheduling (§4.1). Next, we present the design of BIG-C and discuss the changes in YARN to support task preemption (§4.2), and present a preemptive fair share scheduler based on YARN's capacity scheduler (§4.3).

4.1 YARN Resource Management

YARN is a generic resource management framework that allows multiple applications to negotiate resources on a shared cluster. YARN uses *container*, a logical bundle of resources (e.g., ⟨1 CPU, 2GB RAM⟩) as the resource allocation unit. A container is considered as a resource lease and its resources are reclaimed as a whole when a task is completed or killed. The *resource manager* (RM), one per cluster, is responsible for allocating containers to competing applications. The *application master* (AM), one per application, submits requests for containers to RM. The *node manager* (NM), one per machine, monitors the allocation of resources on each node and updates the RM with resource availability.

4.2 BIG-C Design

Figure 4 shows the architecture of BIG-C. The key components of BIG-C include a *resource monitor* (RMon), a preemptive fair scheduler at the RM and a *container allocator* (CA), a *container monitor* (CM) at each NM.

Container allocator Although YARN also uses the notion of "container" in resource management, a YARN container is a logical representation of a task's resources but does not control the actual allocation of resources. The CA component addresses this issue. Upon receiving the request for launching a new task, CA loads the task into a Docker container. Next, CA configures the container with the resources requested by the task.

Container monitor is a per-container daemon in NM responsible for container preemption. Instructed by the NM, CM performs two actions: `container_suspend` and `container_resume` (SR operations in Figure 4). It reconfigures the preempted container to reclaim resources. If memory swapping is detected in a container, CM immediately freezes the container by setting the CPU allocation to 1%.

Resource monitor is a daemon running on the resource manager. It periodically (every 3 seconds) checks the resource distribution among queues according to the current scheduling policy, the resource availability, and resource demands of incoming tasks. Based on the resource sharing policy enforced by the scheduler, RMon together with the *scheduler* compute how much resource should be preempted from over-provisioned queues and send the preemption decision to individual NMs.

4.3 Preemptive Fair Share Scheduler

Overview The preemptive scheduler is built on YARN's capacity scheduler, which enforces fair resource allocation among users. Capacity scheduler is work-conserving and allows users, each assigned with a job queue, to use more than their fair shares if there are available resources in the cluster. When resources are contended, capacity scheduler kills tasks from over-provisioned queues to free resources for under-provisioned queues. The preemptive scheduler replaces kill-based preemption with *immediate preemption* (IP) or *graceful preemption* (GP). While IP does not require algorithmic changes to capacity scheduler, we need to augment the fair sharing algorithm to support GP.

Capacity scheduler's fair sharing algorithm enforces dominant resource fairness (DRF) [13] among job queues. Upon receiving resource requests, in the form of ⟨CPU, RAM⟩, capacity scheduler calculates the dominant resource in these requests and enforces fair allocation of the dominant resource. Non-dominant resources are allocated in proportion to the dominant resource as specified in the requests. Algorithm 1 shows how capacity scheduler calculates the amount of resources to be reclaimed from over-provisioned queues and our modification (highlighted in red) to support GP. For ease of discussion, the algorithm assumes two queues, i.e., one for long jobs and one for short jobs, and two type of resources, i.e., CPU and memory. It can be extended to support more than two queues and two resources.

Capacity scheduler first determines the desired share of resource (line 3). The over-provisioned resources for the long job queue is the difference between the queue's current resource allocation $\vec{r}_l$ and its desired share $\vec{f}_l$ (line 4). If the demand of the short jobs is less than the long job's over-occupied resources, the demand can be fully satisfied (line 5-6). Otherwise, all over-provisioned resources should be reclaimed (line 9). The amount of preempted resources $\vec{p}$ is used in Algorithm 2 to determine

Algorithm 1 Calculate resources to be preempted.

1: **Variables:** Long job's fair share ϕ_l, current resource allocation $\vec{r}_l$, fair allocation $\vec{f}_l$; total CPU C and memory M resources; short job's resource request $\vec{r}_s$; over-provisioned resources $\vec{a}$; resources of long job to be preempted $\vec{p}$.
2: /* Long job's fair and over-provisioned resources */
3: $\vec{f}_l = (C \times \phi_l, M \times \phi_l)$
4: $\vec{a} = \vec{r}_l - \vec{f}_l$
5: **if** $\vec{r}_s < \vec{a}$ **then**
6: $\quad \vec{p} = \vec{r}_s$
7: **else**
8: $\quad$ /* Use DR to calculate preempted resources*/
9: $\quad \vec{p} = \vec{a} \Longrightarrow \vec{p} =$ **ComputeDr**$(\vec{r}_s, \vec{a})$
10: **end if**
11: **procedure** ComputeDr$(\vec{r}_s, \vec{a})$
12: $\quad$ Determine dominant resource
13: $\quad$ **if** dominant resource is CPU **then**
14: $\quad\quad \vec{p} = (a_{cpu}, a_{mem} \times \frac{r_{s_mem}}{r_{s_cpu}})$
15: $\quad$ **else**
16: $\quad\quad \vec{p} = (a_{cpu} \times \frac{r_{s_cpu}}{r_{s_mem}}, a_{mem})$
17: $\quad$ **end if**
18: $\quad$ return $\vec{p}$
19: **end procedure**

Algorithm 2 Container preemption.

1: **Variables:** Set of container to be preempted $\mathbb{C}$; resources to be preempted $\vec{p}$; preempted resources at each GP step $\vec{r}_{GP}$; resources of a preempted container $\vec{r}_c$.
2: AGAIN:
3: Choose a job with the longest remaining time
4: **while** $\vec{p} > (0,0)$ **do**
5: $\quad$ Choose a container c from the job
6: $\quad \mathbb{C} \leftarrow c$
7: $\quad \vec{p} = \vec{p} - \vec{r}_c \Longrightarrow \vec{p} = \vec{p} - \vec{r}_{GP}, \vec{r}_c = \vec{r}_c - \vec{r}_{GP}$
8: $\quad$ remove $c \Longrightarrow$ **if c is empty or swapping, remove c**
9: $\quad$ **if** job has no container left and $\vec{p} > (0,0)$ **then**
10: $\quad\quad$ goto AGAIN
11: $\quad$ **end if**
12: **end while**
13: Kill$(\mathbb{C}) \Longrightarrow$ **Preempt**$(\mathbb{C})$

which containers that belong to the long job should be killed to release these resources.

As shown in Algorithm 2, capacity scheduler uses the heuristic proposed in [37] to choose a job with the longest remaining time and releases $\vec{p}$ resources from its containers. Note that $\vec{p}$ is calculated in algorithm 1. Each time such a container is found, it is added to the kill set $\mathbb{C}$ until either the job has no container left or $\vec{p}$ has been satisfied (line 3-9). If more resources need to be reclaimed, capacity scheduler moves to the next job (line 10). Note that as long as the over-provisioned dominant resource is fully reclaimed, $\vec{p}$ is considered satisfied. The to-be-killed container set $\mathbb{C}$ is then sent to NMs to perform the killings (line 13). There are two drawbacks of capacity scheduler due to kill-based preemption. First, kill-based preemption may lead to resource fragmentation. A killed long job container may be too large for one short task but not sufficient for two short tasks. Second, task killing is not a flexible way to reclaim resource. The killing of a large container only frees resources on one machine and may lead to the launch of a large number of small tasks all clustered on the machine, causing not only load balancing but also reliability problems.

We make simple changes to capacity scheduler to address the above drawbacks. To avoid resource fragmentation, the preempted resource is accurately calculated by function **ComputeDr** based on the demand of short jobs and over-provisioned resources (Algorithm 1, line 9). Specifically, the scheduler computes the dominant resource in request $\vec{r}_s$ against the over-provisioned resource $\vec{a}$. Instead of reclaiming all over-provisioned non-dominant resource as capacity scheduler does, it reclaims the non-dominant resource in proportion to the reclaimed dominant resource as indicated in $\vec{r}_s$. For instance, suppose $\vec{a} = \langle 10 \text{ CPU}, 15\text{GB RAM} \rangle$ and $\vec{r}_s = \langle 20 \text{ CPU}, 10\text{GB RAM} \rangle$. Since $\frac{20}{10} > \frac{10}{15}$, CPU is the dominant resource. Because $\vec{r}_s > \vec{a}$, capacity scheduler will compute $\vec{p} = \vec{a}$ and over-reclaim the memory resource. Instead, the preemptive scheduler computes $\vec{p} = \langle 10 \text{ CPU}, 10 \times \frac{10}{20} \text{ GB RAM} \rangle$. Note that resource preemption based on the dominant resource of short job requests is not possible in the original capacity scheduler because resource allocation of long jobs' containers is based on long jobs' dominant resource and should be reclaimed as a whole.

Further, we introduce graceful task preemption in Algorithm 2. When a job is selected, its over-provisioned resources are reclaimed from a large number of its tasks at a step of $\vec{r}_{GP}$. For each round, the remaining resources of a container c are also updated (line 7). Once a container starts swapping, showing a memory shortage, it is immediately frozen and removed from resource reclamation (line 8). As such, preempted resources are collected from many containers, avoiding drastic slowdown to individual containers as much as possible. As discussed in §2.2, evenly preempting resources from tightly-coupled tasks help minimize slowdowns of Spark jobs.

Practical considerations

Tuning $\vec{r}_{GP}$. The step at which resources are preempted in graceful preemption presents a tradeoff. The larger the $\vec{r}_{GP}$, (hopefully) the sooner the short jobs' demand can be satisfied, but at the risk of causing more pronounced slowdown to long jobs. If $\vec{r}_{GP}$ is too large and pre-

empted containers incur swapping, short jobs can even wait longer for resources to be freed from swapping containers. Small $\vec{r}_{GP}$ leads slow resource allocation to short jobs, which may suffer poor performance after launch.

Delayed resumption factor D. Killed and preempted tasks resubmit their resource requests to the RM and are treated like ordinary incoming tasks. However, resource request from a preempted task has a special locality requirement - it can only be satisfied on the machine where the task was preempted. Moreover, under high burst of short job arrival, a resumed container can be quickly preempted again. The repeated and wasteful preemptions hurt long job performance but also cause long queuing delay to short jobs. To address this issue, we require that a preempted container needs to try D times before it is really resumed. This also avoids possible starvation.

4.4 Implementation

We have implemented BIG-C in Hadoop YARN. The resource monitor is a new module residing in the resource manager that extends `SchedulingEditPolicy`. Our new preemptive fair share scheduler is based on YARN's capacity scheduler. The modifications includes adding a new task state `PREEMPTED`, interfaces for task suspension and resumption in the resource manager. These changes are generic and can interface with any cluster schedulers. On node manager, the container monitor extends `ContainerManagerImpl`. We build a new module called `CoresManager` to handle CPU allocation at worker nodes. A Java interface for `libcontianer` is added to each node manager to operate Docker containers. Our implementation includes 2000 lines of Java code and is based on Hadoop-2.7.1.

5 Evaluation

This section presents the performance evaluation of BIG-C on a 26-node cluster using representative MapReduce and Spark workloads. We first provide details of our testbed (§5.1). Next, we present results from synthetic workloads with MapReduce and Spark jobs (§5.2), and study the sensitivity of two tunable parameters in our design (§5.3). Last, we give results from production workloads based on the Google trace (§5.5).

5.1 Experimental Setup

Cluster Setup Each machine in the 26-node cluster has two 8-core Intel Xeon E5-2640 processors with hyper-threading enabled, 132GB of RAM, and 5x1-TB hard drivers configured as RAID-5. The machines were interconnected by 10 Gbps Ethernet. Hadoop-2.7.1. was deployed on the cluster and HDFS was configured with a replication factor of 3 and a block size of 128MB. The worker nodes sent heartbeats to the resource manager every 3 seconds. Docker-1.12.1 was used to create containers and the images were downloaded from online repository `sequenceiq/hadoop-docker`.

We configured two queues in YARN's resource manager to serve heterogeneous workloads. One queue was dedicated to short jobs and the other was for long jobs. Such a two-queue setting is commonly used in production systems and has been adopted by other works [8, 9, 10]. Additionally, BIG-C can leverage approaches presented in [10, 12] to classify short and long jobs. To enforce strictly higher priority for short jobs, we set the resource share of the short job queue to 95%. The remaining 5% was assigned to the long job queue in a best-effort manner. For comparison, we evaluated the following cluster schedulers:

- **FIFO** scheduler serves all tasks in a single first-in-first-out queue. It achieves optimal performance for long jobs, but incurs significant performance penalty for short jobs.
- **Reserve** schedulers such as Hawk [10] reserve a portion of the cluster to run short jobs exclusively without preemption. Our experiments empirically reserve 60% of cluster resources for short jobs. Long jobs can use up to 40% of cluster capacity. However, it is challenging to find the optimal reservation factor under highly dynamic workloads.
- **Kill** is the preemption mechanism in YARN. The capacity scheduler is used to enforce share between queues. It achieves optimal performance for short jobs, but causes performance degradation to long jobs.
- **IP** and **GP** are immediate preemption and graceful preemptions, respectively. Our preemptive fair share scheduler is used with these two approaches.

Workloads We used Spark-SQL [2] to generate TPC-H queries as short jobs. Hive [30] was used to populate TPC-H tables in HDFS. The total data size was 10GB. The container size for Spark-SQL tasks were set to ⟨4 CPU, 4GB⟩. We selected long jobs from HiBench benchmarks. For MapReduce jobs, we chose map-heavy *wordcount* and reduce-heavy *terasort*. The map and reduce containers were set to ⟨1 CPU, 2GB⟩ and ⟨1 CPU, 4GB⟩, respectively. The input size of the MapReduce jobs was 600GB. We selected *PageRank*, *Kmeans*, *Bayes* and *WordCount* from HiBench as the Spark jobs. The containers of Spark executors were much larger with configurations of ⟨8 CPU, 16GB⟩, ⟨8 CPU, 32GB⟩, and ⟨16 CPU, 32GB⟩, depending on the input size. [3]

Metrics We evaluated the cluster schedulers using the following metrics: *job completion time* (JCT) is the time

[3] The number of CPUs specifies the number of parallel tasks Spark will launch in each executor. Memory size should be large enough to prevent tasks from running into the *Out-Of-Memory* error.

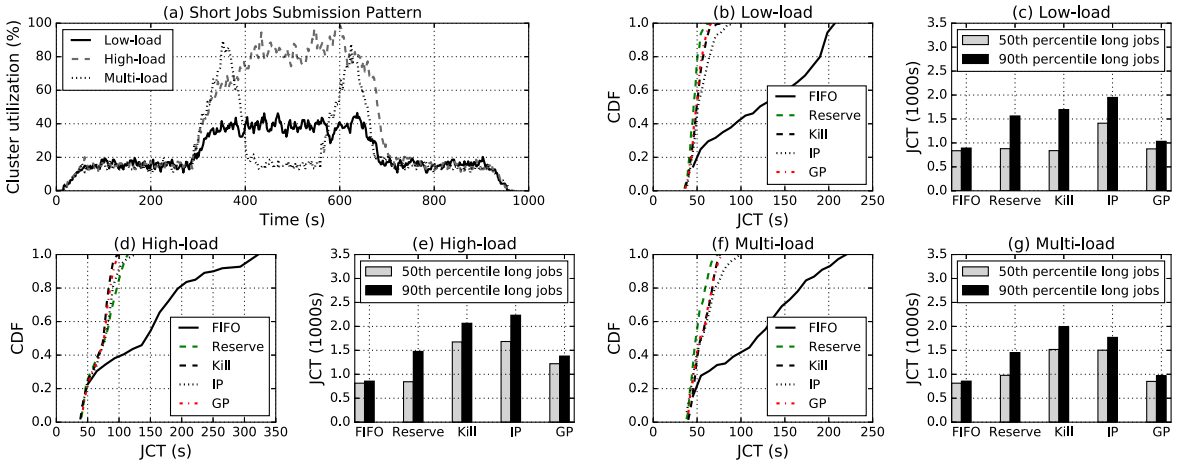


Figure 5: (a) The submission patterns of short jobs in the synthetic workloads. The performance of short Spark-SQL queries (b,d,f) and long Spark jobs (c,e,g) with different schedulers.

when job is submitted until it is completed; *job queueing delay* is the time when a job is submitted until its execution starts; *CoV* of JCT is JCT's coefficient of variation; *cluster utilization* is the CPU utilization over the total CPU capacity of the cluster.

5.2 Results on Synthetic Workloads

Setting In this experiment, we created a controlled environment to study the performance of BIG-C. We generated three workload patterns, each with mixed long and short jobs and lasting for 900 seconds. Long jobs were continuously submitted throughout the experiment and persistently utilized about 80% of the cluster resources. Figure 5(a) shows the submission pattern of short jobs. While all three patterns had a base demand of around 20% cluster capacity, they differed in the submission bursts. *High-load* and *low-load* generated approximately 90% and 40% cluster utilizations, respectively, during the burst period between the 300th and 700th seconds. In contrast, *multi-load* had two peaks during the burst period with each peak demanding over 80% cluster resources. Clearly, the cluster was overloaded during the bursts and long jobs should be preempted by short jobs.

Spark performance Figure 5(b)-(g) shows the results. Among schedulers, *reserve* achieved the best performance for short jobs under low-load and multi-load, since the reserved 60% cluster resources were sufficient to serve the burst. Under high-load, *Reserve* had degraded performance as the resource reservation for short jobs was less than the peak demand. *Kill* was among the best performing schedulers for all three scenarios. In contrast, *FIFO* inflicted substantially delays to short jobs due to the absence of preemption. Short jobs needed to wait for the completion of long jobs before they can be scheduled. Our approaches *IP* and *GP* with the preemptive fair scheduler achieved close performance to the best performing schedulers. Overall, *GP* had superior performance than *IP* as it required less resource reclamation time before short jobs can be launched.

We draw the 50th percentile and 90th percentile performance for long Spark jobs to show the median and the long tail JCTs. *FIFO* achieved the best performance for long jobs because short jobs were unable to preempt and interrupt long job execution. In contrast, *kill* based preemption incurred significant delays to long jobs, especially for the 90th percentile JCT, in all scenarios. It caused on average 140% degradation compared to *FIFO* for the tail JCT. Note that kill-based preemption did not affect the median performance much under low-load since only a few long jobs were killed.

In comparison, *IP* had the worst median and tail performance among all schedulers under low-load. Its aggressive resource preemption and the resulted memory swapping even affected the median JCT under low-load, in which other non-preempted jobs suffered degradation due to memory thrashing. Another reason for *IP*'s poor performance was because complete suspension of one task in Spark jobs often stops the entire job due to task synchronization. Note that even *reserve*, which only dedicated 40% cluster capacity to long jobs, achieved better performance than *kill* and *IP*, indicating that long queuing delay was less damaging on performance than aggressive killings and preemptions. Finally, *GP* was the best performing scheduler compared to baseline *FIFO*. It incurred 13%, 61%, and 17% penalty to the tail performance compared to *FIFO* under low-load, high-load, and multi-load, respectively. The slightly worse median performance of *GP* compared to *kill* was the evidence that many tasks were slowed down due to the collection of preempted resources on these tasks.

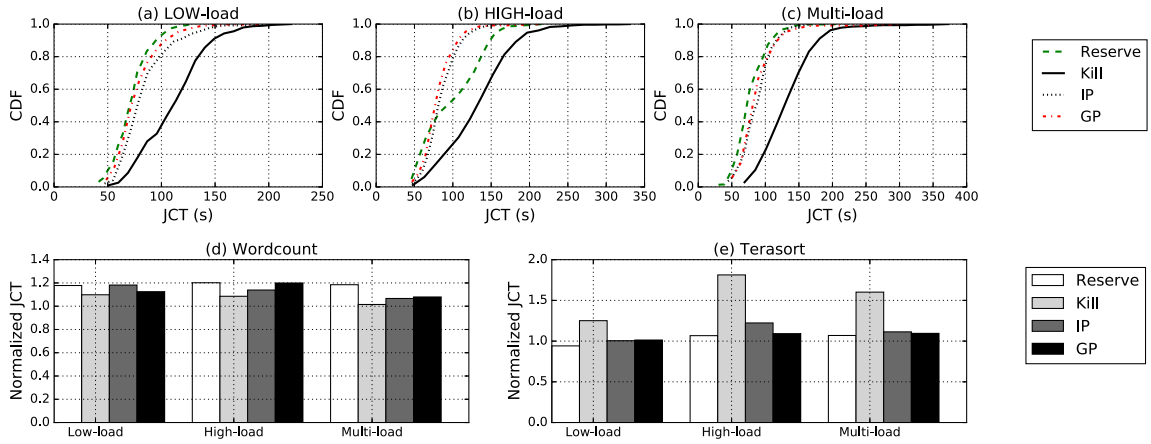


Figure 6: The performance of short Spark-SQL queries (a,b,c) and two long MapReduce jobs (d,e).

Another important finding is that kill-based preemption caused 13%, 15%, and 20% of long jobs failed to complete [4]. In contrast, although *IP* incurred significant overhead, it caused no jobs to fail.

MapReduce performance Figure 6 shows the results of MapReduce workloads. The long job performance is normalized to the scenario in which the cluster is dedicated to long jobs. The short jobs were the same Spark-SQL queries while the long jobs were map-heavy *wordcount* and reduce-heavy *terasort*. MapReduce jobs differ from Spark jobs in many ways. First, a long job usually contains a large number of small mappers, which complete quickly. Second, while Spark's in-memory computing imposes persistent resource demand throughout job execution, MapReduce jobs show clear decline in demand when entering the reduce phase. Finally, MapReduce tasks persist intermediate data onto disk whenever their memory buffers are full. These differences led to different findings in MapReduce workloads.

Figure 6 does not show the results of *FIFO* because the background long jobs had a large number of mappers backlogged and most short jobs suffered 15-20 minutes slowdown. In Figure 6(a)-(c), it is unexpected that *kill* incurred significant queuing delay to short jobs while both *IP* and *GP* performed much better. An examination of YARN's job submission log revealed that the large numbers of killed MapReduce tasks were immediately resubmitted to the scheduler and later killed again, causing wasted cluster resources and additional queuing delays to short jobs. In contrast, both *IP* and *GP* is configured with delayed resumption, which avoided repeated preemptions. *Reserve* had superior performance among the schedulers except for the scenario under high-load, in which the reservation was not sufficient. Both *IP* and *GP* performed well for short jobs.

[4]Failed jobs were not included in JCT calculation.

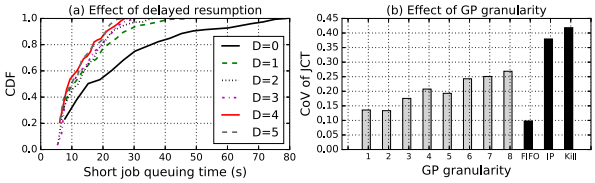


Figure 7: Effects of (a) various degrees of resumption delay and (b) the granularity of graceful preemption.

For long MapReduce jobs, the performance of schedulers depends on the type of the long job workloads. As shown in Figure 6(d), *kill*, *IP*, and *GP* had similar performance for map-heavy workload *wordcount*. It suggests that kill-based preemption is not particularly more expensive than container-based preemption as the mappers are usually small. Because there are a large number of mappers, which are independent from each other, the lost work due to the killings of small mappers can be overlapped with other mappers backlogged in the scheduler. In contrast, kill-based preemption incurred substantial overhead to reduce-heavy *terasort* workload. The cost of killing a reduce task is prohibitively high as relaunching the killed reducer requires re-shuffling all its input data over the network.

Both *IP* and *GP* achieved near-optimal performance with *IP* incurring slight degradation under high-load and multi-load. The write-back of dirty data effectively reduces the in-memory footprint of preempted tasks, making it easier for *IP* and *GP* to reclaim memory compared to that in Spark jobs.

5.3 Parameter Sensitivity

As discussed above, the delayed resumption in our approaches effectively avoided repeated preemptions. In this section, we evaluate the effects of two configurable

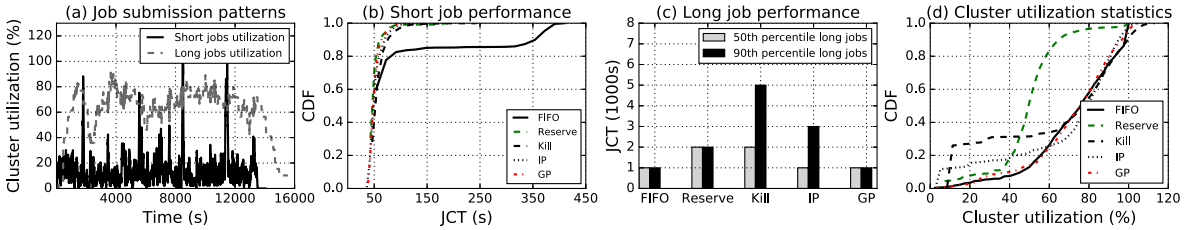


Figure 8: Results on the Google trace. (a) Jobs submission patterns. (b) The CDF of short job's JCT. (c) Long job's 50th and 90th percentile completion time. (d) Statistics on cluster utilization, sampled every 2 seconds.

parameters in our approach. The first parameter is the number of tries a preempted container needs to perform before it is actually resumed. Figure 7(a) shows the effect of varying numbers of the delayed try D on short job performance. The figure suggests that delayed resumption is critical to guaranteeing low latency for short jobs. Disabling delayed resumption ($D = 0$) led to queuing delay as high as 80 seconds for short jobs. Enabling delayed preemption had salient impact on performance but with diminishing gain when further increasing D. We empirically set resumption factor D to 3 to strike a balance between short job latency and long job starvation. This setting was used in all other experiments.

We have shown that there exist tradeoffs between aggressive and graceful resource preemption. Next, we quantitatively study how the granularity (aggressiveness) of GP affects long job performance. We use the coefficient of variance (CoV) of JCT to measure the distribution of preempted resources. The basic preemption unit was set to ⟨1 CPU, 2GB RAM⟩. The GP granularity is determined by how many basic resource units should be reclaimed in one round. Figure 7(b) shows that the CoV of job completion time increased as we increased the granularity of GP. Compared to kill-based and immediate preemptions, graceful preemption, even with aggressive resource reclamation, still incurred less variation across jobs. We set the preemption granularity to ⟨2 CPU, 4GB RAM⟩, i.e., two basic units.

5.4 Overhead

The overhead of BIG-C mainly comes from reclaiming the memory of preempted tasks and the delay caused by memory restoration. Our experiments show that it takes approximately 3 seconds to reclaim 1GB dirty memory, which adds considerable scheduling delay to short jobs. Although BIG-C avoids such overhead for most of time due to graceful preemption, performance degradation is inevitable if GP fails to satisfy short job demands.

5.5 Results on the Google Trace

We also evaluated BIG-C by replaying the production Google trace on our testbed. This trace contains 2202 jobs, of which 2020 are classified as short jobs and 182 as long jobs based on job completion time and resource usage. The setting conforms with that used in [10]. We scaled down the task numbers in each job to match our cluster capacity so that each job takes a reasonable amount of time to complete. The total trace ran for 3.8 hours. We first dedicated the entire cluster to short jobs and long jobs to respectively quantify their resource usage. The results are shown in Figure 8(a). The average cluster utilization was about 17% and 75% for short and long jobs, respectively. The short jobs only consumed a small portion of the total resource, but with highly variable and unpredictable submission rates. We can clearly see a few short job usage spikes throughout the trace. For example, the spike at the 8000th second used up to 95% of the cluster capacity. Similarly, we configured the short job share to be 95% of the cluster capacity.

Figure 8(b) plots the latency distribution of short job in the Google trace. Most schedulers except *FIFO* achieved good performance. *FIFO* had a 90th percentile latency of 335s, which was 6 times larger than its median latency. We also examined the tail latency under other schedulers. The 95th percentile latency for *reserve*, *kill*, *IP*, and *GP* were 183s, 176s, 118s, and 96s, respectively. Our two approaches outperformed other schedulers with *GP* being the best.

Figure 8(c) draws the 50th percentile and 90th percentile performance for long jobs. With the default kill-based preemption, 105 out of the 182 long jobs were killed, among which 41 failed and 105 suffered significant slowdown due to job re-launch. Note that failed jobs were excluded from JCT calculation. As shown in the figure, *GP* improved the 90th percentile job runtime by 67%, 37% and 32% over *kill*, *IP*, and *reserve*, respectively. Compared to the optimal *FIFO* scheduler for long jobs, *GP* only added 4% delay to JCT. Similarly, about 23% long jobs failed with kill-based preemption while our approaches did not cause any job failures.

Figure 8(d) plots the cluster utilization under different schedulers. Work-conserving schedulers achieved much higher resource utilizations than *reserve* did. For more than 60% of time in the experiment, the overall cluster

utilization was above 80% for *FIFO*, *kill*, *IP*, and *GP*. In contrast, *reserve* rarely used more than 60% of cluster capacity due to the reservation for short jobs. Note that both *kill* and *IP* had periods when the cluster utilization was lower than 40%. This was due to the killing and aggressive preemption of tasks that impeded the overall progress of the tightly-coupled long jobs, e.g., Spark jobs. When waiting for the killed or preempted task to relaunch or resume, other sibling tasks were idling.

6 Related Work

The last few years have witnessed the growth of workloads provisioned on top of data processing frameworks like Hadoop [1], Naiad [23] or Spark [36]. The characteristics of such workloads have been well-studied in previous work [32].

Cluster Scheduling is a core component in data-intensive cluster computing. YARN [31] and Mesos [16] are two widely used open-source cluster managers. Both YARN and Mesos use a two-level architecture, decoupling allocation from application-specific logic such as task scheduling, speculative execution or failure handling. Omega [29] is a parallel scheduler architecture based on lock-free optimistic concurrency control to achieve implementation extensibility, globally optimized scheduling, and scalability. Another thread of work focuses on distributed scheduler to overcome the scalability problem in large-scale clusters. Sparrow [25] is a fully distributed scheduler that performs scheduling by performing randomized sampling. Hawk [10] and Mercury [18] both implement a hybrid scheduler to avoid inferior scheduling decisions for a subset of jobs as a trade-off of scheduling quality and scalability. yaq-d and yaq-c [27] provides queue management at worker nodes to improve cluster utilization and to avoid head-of-line blocking. Our proposed container-based preemption is orthogonal to these approaches and helps simplify the design of cluster schedulers by providing an alternative means of enforcing task priority. Note that our work does not intend to improve task classification but focuses on a lightweight mechanism for task preemption.

Preemption Amoeba [3] and Natjam [8] implement preemption using checkpointing to achieve elastic resource allocation. Natjam targets at Hadoop applications and Amoeba built a prototype based on Sailfish [26]. Li et. al., propose a new checkpoint mechanism by leveraging CRIU [19]. Their approaches interact with the Application Master and dump the checkpoints to user space. There are two drawbacks in checkpoint-based preemption. First, it is challenging to determine the frequency of checkpointing. On-demand checkpointing, such as the preemption approaches based on CRIU [19], requires the entire preempted task to be dumped onto HDFS and is equivalent to our proposed immediate preemption. Periodic checkpointing at each iteration reduces preemption delay but incurs considerable runtime overhead. Second, existing checkpointing approaches require changes to user applications. Our proposed container-based preemption is application agnostic and the tuning of the GP granularity is relatively straightforward.

Utilization To improve cluster utilization, authors in [21, 35, 20, 38, 14] propose to consolidate applications on a shared infrastructure and separately manage their interference so that applications' QoS could be guaranteed. These techniques employ online profiling to identify the best combinations of workloads that do not interfere with each other. However, in data center scheduling, in which job submissions are unpredictable and the composition of jobs is heterogeneous, offline training or online profiling may not be accurate. Our approach does not require the cluster to be under-provisioned nor assumes scheduling-friendly job submissions.

Lightweight virtualization Container-based virtualization have been widely used both in industry and in research. Xavier et. al., [34] evaluated the HPC performance in container based environments. Burns et al., [6] propose a new design pattern for container based distributed systems. Google Borg [33] has used OS container to aid cluster management. However, its container usage is limited to task isolation and preemption is still based on task killing. Harter et al., [15] propose a Docker storage driver to enable fast container startup. The YARN community has also provided support to run Docker containers to replace the logical YARN container. However, there still lacks support to fully control the resource allocation to containers in YARN.

7 Conclusion

In this paper, we tackle the problem of scheduling heterogeneous workloads on a shared cluster. Inspired by task scheduling in operating systems, in which fast and low-cost preemption is key to achieving both responsiveness and high utilization, we leverage lightweight virtualization to enable task preemption in cluster computing, such as YARN. Experimental results show that our proposed mechanism for preemption is effective for different types of Big Data workloads, e.g., MapReduce and Spark. Note that container-based preemption is not yet suitable for workloads with sub-second latency, like those studied in [25]. Suspending and saving the context of a data-intensive task still takes a few seconds. Providing extremely low-latency task preemption for sub-second workloads is an interesting future direction.

Acknowledgement We are grateful to our reviewers for their comments on this paper and our shepherd Mona Attariyan for her suggestions. This research was supported in part by U.S. NSF grants CNS-1422119, CNS-1649502 and IIS-1633753.

References

[1] Apache hadoop project. `https://hadoop.apache.org/`.

[2] Spark-sql. `http://spark.apache.org/sql/`.

[3] ANANTHANARAYANAN, G., DOUGLAS, C., RAMAKRISHNAN, R., RAO, S., AND STOICA, I. True elasticity in multi-tenant data-intensive compute clusters. In *Proceedings of the Third ACM Symposium on Cloud Computing* (2012).

[4] ARON, M., DRUSCHEL, P., AND ZWAENEPOEL, W. Cluster reserves: a mechanism for resource management in cluster-based network servers. In *Proceedings of ACM SIGMETRICS Performance Evaluation Review* (2000).

[5] BARROSO, L. A., AND HOELZLE, U. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009.

[6] BURNS, B., AND OPPENHEIMER, D. Design patterns for container-based distributed systems. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)* (2016).

[7] CHEN, Y., ALSPAUGH, S., AND KATZ, R. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *In Proceedings of the VLDB Endowment* (2012).

[8] CHO, B., RAHMAN, M., CHAJED, T., GUPTA, I., ABAD, C., ROBERTS, N., AND LIN, P. Natjam: Design and evaluation of eviction policies for supporting priorities and deadlines in mapreduce clusters. In *Proceedings of the 4th annual Symposium on Cloud Computing* (2013).

[9] 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* (2014).

[10] DELGADO, P., DINU, F., KERMARREC, A.-M., AND ZWAENEPOEL, W. Hawk: Hybrid datacenter scheduling. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC 15)* (2015).

[11] 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).

[12] 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).

[13] GHODSI, A., ZAHARIA, M., HINDMAN, B., KONWINSKI, A., SHENKER, S., AND STOICA, I. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation* (2011).

[14] GRANDL, R., CHOWDHURY, M., AKELLA, A., AND ANANTHANARAYANAN, G. Altruistic scheduling in multi-resource clusters. In *Proceedings of OSDI16: 12th USENIX Symposium on Operating Systems Design and Implementation* (2016).

[15] HARTER, T., SALMON, B., LIU, R., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Slacker: fast distribution with lazy docker containers. In *Proceedings of 14th USENIX Conference on File and Storage Technologies (FAST 16)* (2016).

[16] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R. H., SHENKER, S., AND STOICA, I. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation* (2011).

[17] HUANG, S., HUANG, J., DAI, J., XIE, T., AND HUANG, B. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Proceedings of the Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on* (2010).

[18] KARANASOS, K., RAO, S., CURINO, C., DOUGLAS, C., CHALIPARAMBIL, K., FUMAROLA, G. M., HEDDAYA, S., RAMAKRISHNAN, R., AND SAKALANAGA, S. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC 15)* (2015).

[19] LI, J., PU, C., CHEN, Y., TALWAR, V., AND MILOJICIC, D. Improving preemptive scheduling with application-transparent checkpointing in shared clusters. In *Proceedings of the 16th Annual Middleware Conference* (2015).

[20] LO, D., CHENG, L., GOVINDARAJU, R., RANGANATHAN, P., AND KOZYRAKIS, C. Heracles: improving resource efficiency at scale. In *Processings of the ACM SIGARCH Computer Architecture News* (2015).

[21] 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).

[22] MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. *Proceedings of the Linux Journal* (2014).

[23] 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).

[24] NGUYEN, K., FANG, L., XU, G., DEMSKY, B., LU, S., ALAMIAN, S., AND MUTLU, O. Yak: A high-performance big-data-friendly garbage collector. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016).

[25] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013).

[26] RAO, S., RAMAKRISHNAN, R., SILBERSTEIN, A., OVSIANNIKOV, M., AND REEVES, D. Sailfish: A framework for large scale data processing. In *Proceedings of the Third ACM Symposium on Cloud Computing* (2012).

[27] RASLEY, J., KARANASOS, K., KANDULA, S., FONSECA, R., VOJNOVIC, M., AND RAO, S. Efficient queue management for cluster scheduling. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016).

[28] REISS, C., TUMANOV, A., GANGER, G. R., KATZ, R. H., AND KOZUCH, M. A. Towards understanding heterogeneous clouds at scale: Google trace analysis. *Proceedings of the Intel Science and Technology Center for Cloud Computing, Tech. Rep* (2012).

[29] SCHWARZKOPF, M., KONWINSKI, A., ABD-EL-MALEK, M., AND WILKES, J. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013).

[30] 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. *Proceedings of the VLDB Endowment* (2009).

[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).

[32] 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)* (2016).

[33] 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).

[34] XAVIER, M. G., NEVES, M. V., ROSSI, F. D., FERRETO, T. C., LANGE, T., AND DE ROSE, C. A. Performance evaluation of container-based virtualization for high performance computing environments. In *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (2013).

[35] YANG, H., BRESLOW, A., MARS, J., AND TANG, L. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the ACM SIGARCH Computer Architecture News* (2013).

[36] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. *Proceedings of HOTCLOUD'16 USENIX Workshop on Hot Topics in Cloud Computing* (2010).

[37] ZAHARIA, M., KONWINSKI, A., JOSEPH, A. D., KATZ, R. H., AND STOICA, I. Improving mapreduce performance in heterogeneous environments. In *OSDI* (2008).

[38] ZHANG, Y., PREKAS, G., FUMAROLA, G. M., FONTOURA, M., GOIRI, Í., AND BIANCHINI, R. History-based harvesting of spare cycles and storage in large-scale datacenters. In *Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016).

The RCU-Reader Preemption Problem in VMs

Aravinda Prasad
Indian Institute of Science, Bangalore

K. Gopinath
Indian Institute of Science, Bangalore

Paul E. McKenney
IBM Linux Technology Center, Beaverton

Abstract

When synchronization primitives such as locking and read-copy update (RCU) execute within virtual machines (VMs), preemption can cause multi-second latency spikes, increasing peak memory footprint and fragmentation inside VMs, which in turn may trigger swapping or VM ballooning. The resulting CPU utilization and memory footprint increases can negate the server-consolidation benefits of virtualization. Although preemption of lock holders in VMs has been well-studied, the corresponding solutions do not apply to RCU due to its exceedingly lightweight read-side primitives.

This paper presents the first evaluation of RCU-reader preemption in a virtualized environment. Our evaluation shows 50% increase in the peak memory footprint and 155% increase in fragmentation for a microbenchmark, 23.71% increase in average kernel CPU utilization, 2.9× increase in the CPU time to compute a grace period and 2.18× increase in the average grace period duration for the Postmark benchmark.

1 Introduction

Virtualization brings server-consolidation benefits to the cloud environment by multiplexing physical resources across virtual machines (VMs), but can lead to problematic preemption. For example, preemption of the virtual CPU (vCPU) holding a lock can cause latency spikes [18] because other vCPUs continue spinning to acquire the lock until the lock-holder vCPU resumes.

Well-known solutions to lock-holder preemption include priority inheritance [16, 8], and more recent work proposes solutions for the preemption of vCPUs holding locks [18, 14, 17, 2, 20, 23, 21]. Unfortunately, the heavyweight solutions proposed for lock-holder vCPU preemption, such as priority inheritance, do not apply to RCU because (1) RCU's read-side primitives must be exceedingly lightweight, and (2) preemption of RCU readers provokes different failure modes such as increased memory footprint. Nevertheless, preemption of vCPUs executing RCU readers has received little attention.

To the best of our knowledge, this is the first evaluation of vCPU preemption within RCU readers.

2 The RCU synchronization technique

Read-Copy-Update (RCU) [9, 12, 13] is a highly scalable structured-deferral [11] synchronization technique. *RCU read-side critical sections* are bounded by `rcu_read_lock()` and `rcu_read_unlock()`, which are bounded population-oblivious wait-free primitives that need not directly synchronize with writers. In consequence, each writer must guarantee that all data structures may be safely traversed by readers at all times.

For example, a writer deleting an object from a linked list first removes the object, then uses `synchronize_rcu()` to wait for all pre-existing readers to finish. Because new readers cannot gain a reference to the newly removed object, once all pre-existing readers complete, only the writer will have a reference to that object, which can then be safely freed. This writer-wait time period is called an *RCU grace period* (GP). Writers that cannot block may instead use `call_rcu()`, which posts an *RCU callback* that invokes a specified function with a specified argument after the completion of a subsequent GP. Although GPs can be expensive, batching optimizations allow thousands of `synchronize_rcu()` and `call_rcu()` requests to share a single GP [15], resulting in extremely low per-request GP overhead.

While the RCU-reader preemption problem is applicable across all RCU variants, this paper focuses on the "classic" RCU used by server builds of the Linux kernel. The "classic" RCU prohibits readers from executing any sort of context switch, as is also prohibited for spinlock holders. Therefore, any time interval during which all CPUs execute a context switch is by definition an RCU GP, as illustrated by Figure 1 [9, 12].

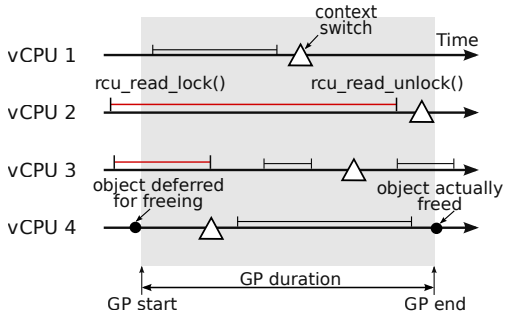


Figure 1: Linux-kernel grace period (GP). Red critical sections marked might hold references to the deferred object.

3 The RCU-reader preemption problem

RCU GPs cannot complete while a vCPU is preempted within an RCU read-side critical section. Thus, calls to `synchronize_rcu()` cannot return, and although calls to `call_rcu()` continue to return immediately, their callbacks cannot be invoked. Linux-kernel code can therefore continuously invoke `call_rcu()`, resulting in an unbounded quantity of memory that cannot be reused until the GP completes.

For example, consider an RCU-protected hash table that is searched incessantly and updated frequently, with deletions invoking `call_rcu()` to safely free old hash-table elements after a GP has elapsed. Suppose that just one vCPU is preempted within an RCU read-side critical section, but that the other vCPUs continue execution unhindered. These other vCPUs will continue their reads and updates, but because GPs cannot complete, elements deleted from the hash table cannot be freed until the preempted vCPU resumes its execution. This will increase memory footprint, which can in turn increase CPU utilization, for example, due to increased numbers of cache and TLB misses. CPU utilization can also increase because RCU takes increasingly aggressive measures in an attempt to force the preempted vCPU to execute the context switch needed to allow GP to complete. Unfortunately, these measures are futile because the vCPU itself has been preempted.

The RCU-reader preemption vs lock-holder preemption: While the usual symptom of lock-holder preemption is to hang all or part of the system, RCU-reader preemption instead bloats memory footprints.

Techniques to handle lock-holder preemption such as preemption-aware scheduling [23, 21] make the hypervisor aware of lock contention within the guest, and can be augmented by hardware support [20]. For instance, Intel's hardware-based Pause-Loop Exiting feature can detect a vCPU spinning on a lock. However, these techniques cannot be applied directly to RCU because RCU's server-build read-side primitives do not make any state change detectable by hypervisor or hardware (in fact the RCU's server-build read-side primitives are a *no-op* [10]). Although read-side primitives could make such a state change, doing so is problematic for two reasons. First, RCU's primary goal is zero or low-overhead read-side primitives, so RCU must push such overheads to writers. Second, state-change overheads are unacceptable for read-only or read-mostly data structures tracking the systems hardware configuration (e.g., active disks and online CPUs) where the read-to-write ratio (e.g., accessing a disk to replacing a disk) is well in excess of ten to the ninth power.

Therefore, alternative approaches are required to handle the RCU-reader preemption problem.

4 Impact of RCU-reader preemption

In this section we discuss both primary and secondary impacts due to the RCU-reader preemption problem.

Latency: Guest OSes invoking `synchronize_rcu()` can incur latency spikes of several *seconds* on overcommitted hosts. These spikes' durations depend directly on the vCPU preemption time.

Transient memory spikes: As discussed earlier, when using `call_rcu()`, GP delay due to vCPU preemption can cause transient memory-footprint spikes, which can in turn increase peak memory footprint.

Fragmentation inside VMs: Frequent transient memory-footprint spikes can scatter the kernel pages throughout the system, which can increase external memory fragmentation [4]. This fragmentation can cause premature memory-allocation failure, especially for hugepage allocations.

Swapping and Ballooning: Cloud environments often provision memory on an as-needed basis in order to reduce memory costs. Increased peak-memory footprint can trigger swapping, degrading performance and generating additional I/O load.

Furthermore, some cloud service providers oversubscribe memory because VMs do not always consume all their memory [22]. The combination of memory-footprint spikes and oversubscription can cause balloon drivers [19] to be frequently invoked as the hypervisor reacts to these spikes, further increasing overhead.

CPU utilization: The above issues can increase CPU utilization. For example, fragmentation might trigger compaction, which can consume significant CPU time while scanning and migrating memory.

VM density and consolidation: Increased peak-memory footprint require VMs to be provisioned with more memory, degrading VM density and consolidation, in turn increasing costs and energy utilization.

5 Factors influencing the impact of RCU-reader preemption

vCPU preemption time: GP-completion delays depend on vCPU preemption duration, which in turn depends on the hypervisor's CPU overcommit factor; higher overcommit factors increase vCPU preemption frequency which increases GP-completion delays.

RCU read-side critical section length: GP duration depends on read-side critical-section duration which, in the non-preemptible kernels this paper focuses on, depends on the time between voluntary context switches. As a rule of thumb, the longer this time, the greater the probability of preemption, and thus the greater the probability of GP-completion delays.

Objects allocation and defer free rate: Given vCPUs being preempted within RCU read-side critical sections, workloads that invoke `call_rcu()` frequently will see larger memory-footprint spikes than workloads that instead use `synchronize_rcu()`. Of the workloads that invoke `call_rcu()` frequently, those that allocate larger blocks of memory will see correspondingly larger memory-footprint spikes.

Total kernel time: Compute-intensive workloads spend little time in the kernel, which in turn means a given vCPU spends little time executing in-kernel RCU read-side critical sections. Therefore, RCU-reader preemption has a smaller effect on these workloads.

6 Evaluation

We evaluate a mail server benchmark, a memory-allocator intensive microbenchmark and a namespace cloning microbenchmark to understand the RCU-reader preemption impact under different stress conditions.

6.1 Benchmarks

Postmark [5] simulates a mail server's file create, delete, read and write operations. We run the benchmark on an in-memory filesystem starting with 128K files.

Memory microbenchmark, implemented as a kernel module, allocates an object of size 1K followed by a call to `call_rcu()` to reclaim the object after a GP.

Clone microbenchmark measures how quickly a new namespace can be cloned by calling the `clone()` system call in a loop from a user space program. Namespace cloning, for example, is employed by chroot jailing to create filesystem-isolated processes [6] and also in web server security that places the per user worker process into an isolated network [1].

6.2 Test setup

The host is an Intel Xeon E5-4640 processor having 64 CPUs (4 CPU sockets, 8 cores per socket and two-way hyper-threading) and 236 GB of physical memory. The host uses KVM [7] virtualization under Linux kernel 4.5.0 for both host and guests. Baseline measurements boot only the VM running the benchmark.

Experiment 1:

Instance	vCPUs	CPU Affinity	Memory
VM1	32	0–31	8 GB
VM2	32	32–63	8 GB
VM3	8	0–31	4 GB
VM4	8	32–63	4 GB

VM1 runs Postmark benchmark with 32 instances. VM2 runs memory microbenchmark with 32 parallel kernel threads. Both VM3 and VM4 run a bursty workload with 8 user space process. The bursty workload randomly executes 0.1 to 10 million arithmetic operations followed by randomly sleeping for 1 to 200 milliseconds.

Experiment 2:

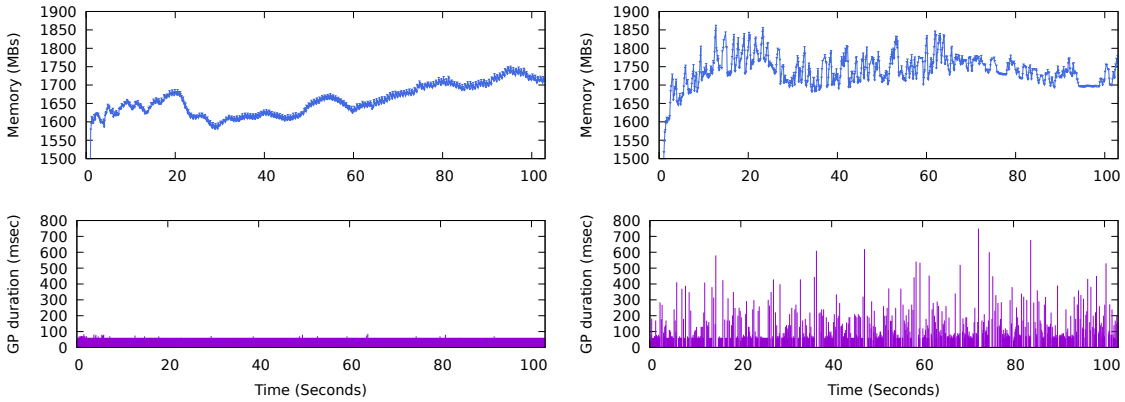
Instance	vCPUs	CPU Affinity	Memory
VM1	64	0–63	8 GB
VM2	16	0–31	8 GB

VM1 runs the clone microbenchmark and VM2 runs 16 CPU-hogging user processes on 32 vCPUs.

6.3 Results

Postmark: The file create and delete operations issued by the Postmark benchmark allocate filesystem objects such as `inode` and `dentry` (directory entry), and delete the objects by invoking `call_rcu()`. While the reclamation of the deferred objects' memory is delayed due to longer GPs, other benchmark threads continue performing file creation and deletion resulting in increased memory footprint.

Figure 2 reveals memory-footprint spikes in overcommit scenario due to delayed reclamation of `inode` and `dentry` objects. The vCPU preemption induces longer GPs which in turn delays the reclamation of deferred objects. There are no spikes in the baseline scenario because timely GP completion results in timely reclamation of memory.



(a) Baseline (b) Overcommit

Figure 2: Memory trace and GP durations for the first 100 seconds of the Postmark benchmark execution

Description	Baseline	Overcommit
Mean GP duration (ms)	57.6 (± 10.8)	125.9 (± 114.3)
Max GP duration (ms)	89.93	2372.12
Min GP duration (ms)	5.62	4.32
90th %-tile (ms)	60.05	251.36
50th %-tile (ms)	59.99	80.18
CPU consumed per GP (μs)	633.85	1833.54

Table 1: GP statistics for the Postmark benchmark

Description	Baseline	Overcommit
Mean GP duration (ms)	53.27 (± 13.4)	69.39 (± 30.4)
Max GP duration (ms)	87.66	317.59
Min GP duration (ms)	8.88	9.13
90th %-tile (ms)	60.18	109.98
50th %-tile (ms)	59.94	60.32
CPU consumed per GP (μs)	860.26	1095.72

Table 2: GP statistics for the memory microbenchmark

Table 1 shows a 2.18× increase in the average GP duration due to 6.33× increase in the number of RCU-reader preemption events extending GP duration. RCU's aggressive context-switch forcing results in a 2.9× increase in GP-computation time and further contributes to a 23.71% increase in kernel CPU utilization on overcommitted hosts.

Scattering of kernel pages due to frequent memory-footprint spikes results in a 32.5% increase in external fragmentation (computed using the `debugfs` "unusable free space index" for huge page allocations [3]) during benchmark execution when the host is overcommitted.

The above factors contribute to a 66.73% decrease in the throughput of the Postmark benchmark. However, the throughput is also affected by other factors including increased context-switch rates, preemption of vCPU holding a spinlock and reduction in number of vCPU assigned to the VM during host overcommit. We are currently investigating how much of this throughput decrease is due to RCU-reader preemption.

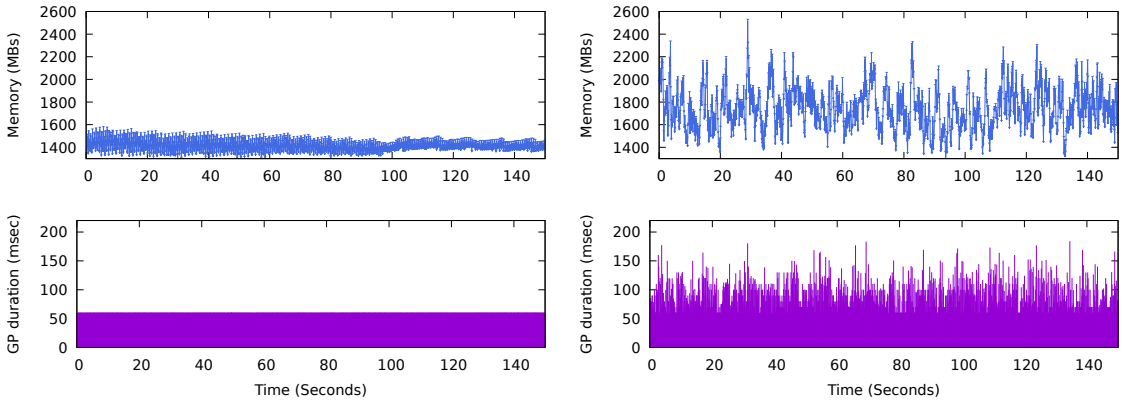
Memory microbenchmark: We run a memory-allocator-intensive benchmark to evaluate and understand the impact of RCU-reader preemption on GP durations and memory-footprint spikes. The microbenchmark issues 2.5K pairs of allocations and `call_rcu()` invocations per second per CPU. It also invokes the scheduler after every ten allocation-`call_rcu()` pairs to limit the duration of the resulting RCU read-side critical sections.

Figure 3 shows memory-footprint spikes of several hundred MBs due to longer GPs when the host is overcommitted. The resulting RCU-reader preemption results in a 50% increase in the peak memory footprint (and an 842 MB increase in peak memory footprint), a 30.26% increase in the average GP duration (Table 2) and a 155.32% increase in external fragmentation.

This microbenchmark shows a significant memory-footprint sensitivity to GP duration: A short 100-millisecond GP delay results in spikes of several hundred MBs in the memory footprint. In contrast, the Postmark benchmark, with its lower `call_rcu()` frequency, has a smaller memory-footprint sensitivity to GP duration, so that a longer 400-millisecond GP delay results in a memory-footprint spike of only about 50-100 MB.

Clone microbenchmark: The `clone` system call allocates several kernel objects during namespace cloning which are passed to `call_rcu()` when the last process exits that namespace. The `clone` microbenchmark therefore repeatedly invokes `clone` in a loop.

Figure 4 reveals occasional spikes in GP duration inside the VM running clone microbenchmark, even when the host's average CPU utilization is 28%. Such spikes depend on the vCPU preemption timing and result in



(a) Baseline (b) Overcommit

Figure 3: Memory trace and GP durations for the first 150 seconds of the memory microbenchmark execution

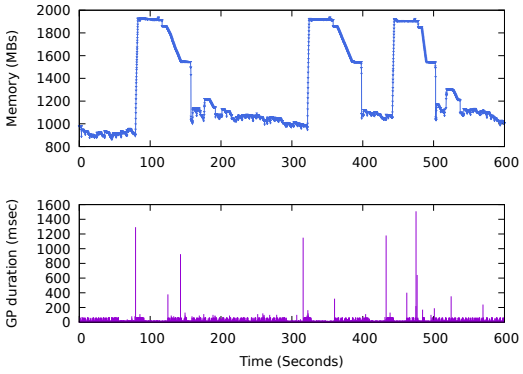


Figure 4: Memory trace and GP duration for the clone microbenchmark when the host is overcommitted

1 GB memory-footprint spikes persisting for several seconds. This result means that adding VMs (thus increasing the rate of `clone` invocations) can have the counterproductive effect of disproportionately increasing memory footprint due to increased RCU-reader preemption.

7 Discussion

RCU-reader preemption on an overcommitted host can result in latency spikes, increasing peak memory footprint and fragmentation within VMs. These increases can in turn increase CPU utilization due to increases in cache and TLB misses and due to additional memory-compaction operations. This increase in CPU utilization can reduce or even negate the cost and energy-efficiency benefits of server consolidation.

Cloud service providers and VM users should consider host overcommit ratios and workload sensitivities to delayed GPs while provisioning VM resources. Although GP-sensitive workloads can be identified via kernel profiling of `call_rcu()` and `synchronize_rcu()` invocations, it is currently difficult to determine the required changes to per-VM resource provisioning.

Furthermore, given systems with CPU overcommit, a CPU-consumption spike in one VM might cause a GP-duration spike in another VM. This sort of cross-VM interaction poses significant challenges for VM resource provisioning, which further motivates an effective solution to the problem of preemption of vCPUs running RCU read-side critical sections.

We are therefore currently investigating a holistic solution for the RCU-reader preemption problem that combines changes to the Linux-kernel RCU implementation, the guest-OS memory allocator, the hypervisor scheduler and the subsystems using RCU. The solution aims to reduce the GP duration on overcommitted hosts.

8 Conclusion

This paper introduces the RCU-reader vCPU preemption problem and demonstrates that it has significant and far-reaching performance impacts. We are investigating potential solutions to this problem.

9 Acknowledgments

We thank our shepherd, Eddie Kohler, and the anonymous reviewers for their helpful comments.

Disclaimer: The views in the article are solely of the authors and not of their employers.

References

[1] EDGE, J. Namespaces in operation, part 7: Network namespaces. `https://lwn.net/Articles/580893/`, 2014.

[2] FRIEBEL, T., AND BIEMUELLER, S. How to deal with lock holder preemption. *Xen Summit North America* (2008).

[3] GORMAN, M. mm: Export unusable free space index via debugfs. `https://lkml.org/lkml/2010/4/20/307`, 2010.

[4] GORMAN, M., AND WHITCROFT, A. The what, the why and the where to of anti-fragmentation. In *Ottawa Linux Symposium* (2006), vol. 1, Citeseer, pp. 369–384.

[5] KATCHER, J. Postmark: A new filesystem benchmark. Tech. rep., Technical Report TR3022, Network Appliance, 1997.

[6] KERRISK, M. Namespaces in operation, part 1: namespaces overview. `https://lwn.net/Articles/531114/`, 2013.

[7] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium* (2007), vol. 1, pp. 225–230.

[8] MCKENNEY, P. Priority-Boosting RCU Read-Side Critical Sections. `https://lwn.net/Articles/220677`, 2007.

[9] MCKENNEY, P. E. *Exploiting Deferred Destructions: An Analysis of Read-Copy-Update Techniques in Operating System kernels*. PhD thesis, Oregon Health & Science University, 2004.

[10] MCKENNEY, P. E. What is RCU? Part 2: Usage. `http://lwn.net/Articles/263130/`, 2007.

[11] MCKENNEY, P. E. Structured deferral: synchronization via procrastination. *Commun. ACM 56*, 7 (July 2013), 40–49.

[12] MCKENNEY, P. E., APPAVOO, J., KLEEN, A., KRIEGER, O., RUSSELL, R., SARMA, D., AND SONI, M. Read-Copy Update. In *AUUG Conference Proceedings* (2001), AUUG, Inc., p. 175.

[13] MCKENNEY, P. E., AND SLINGWINE, J. D. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems* (1998), pp. 509–518.

[14] OUYANG, J., AND LANGE, J. R. Preemptable ticket spinlocks: Improving consolidated performance in the cloud. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 191–200.

[15] SARMA, D., AND MCKENNEY, P. E. Making RCU safe for deep sub-millisecond response realtime applications. In *Proceedings of the 2004 USENIX Annual Technical Conference (FREENIX Track)* (2004), pp. 182–191.

[16] SHA, L., RAJKUMAR, R., AND LEHOCZKY, J. P. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers 39*, 9 (1990), 1175–1185.

[17] SUKWONG, O., AND KIM, H. S. Is co-scheduling too expensive for SMP VMs? In *Proceedings of the Sixth Conference on Computer Systems* (2011), ACM, pp. 257–272.

[18] 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.

[19] WALDSPURGER, C. A. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 181–194.

[20] WELLS, P. M., CHAKRABORTY, K., AND SOHI, G. S. Hardware support for spin management in overcommitted virtual machines. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques* (2006), ACM, pp. 124–133.

[21] WENG, C., LIU, Q., YU, L., AND LI, M. Dynamic adaptive scheduling for virtual machines. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing* (2011), ACM, pp. 239–250.

[22] WILLIAMS, D., JAMJOOM, H., LIU, Y.-H., AND WEATHERSPOON, H. Overdriver: Handling memory overload in an oversubscribed cloud. In *ACM SIGPLAN Notices* (2011), vol. 46, ACM, pp. 205–216.

[23] ZHANG, L., CHEN, Y., DONG, Y., AND LIU, C. Lock-visor: An efficient transitory co-scheduling for MP guest. In *Proceedings of the 2012 41st International Conference on Parallel Processing* (Washington, DC, USA, 2012), ICPP '12, IEEE Computer Society, pp. 88–97.

BUNSHIN: Compositing Security Mechanisms through Diversification

Meng Xu, Kangjie Lu, Taesoo Kim, Wenke Lee
Georgia Institute of Technology

Abstract

A number of security mechanisms have been proposed to harden programs written in unsafe languages, each of which mitigates a specific type of memory error. Intuitively, enforcing multiple security mechanisms on a target program will improve its overall security. However, this is not yet a viable approach in practice because the execution slowdown caused by various security mechanisms is often non-linearly accumulated, making the combined protection prohibitively expensive; further, most security mechanisms are designed for independent or isolated uses and thus are often in conflict with each other, making it impossible to fuse them in a straightforward way.

In this paper, we present BUNSHIN, an N-version-based system that enables different and even conflicting security mechanisms to be combined to secure a program while at the same time reducing the execution slowdown. In particular, we propose an automated mechanism to distribute runtime security checks in multiple program variants in such a way that conflicts between security checks are inherently eliminated and execution slowdown is minimized with parallel execution. We also present an N-version execution engine to seamlessly synchronize these variants so that all distributed security checks work together to guarantee the security of a target program.

1 Introduction

Memory errors in programs written in unsafe languages (e.g., C/C++) have been continuously exploited by attackers [14]. To defeat such attacks, the security community has deployed many security mechanisms such as widely deployed W⊕X, which prevents code injection attacks by making Writable memory not eXecutable, and ASLR, which prevents attacks (e.g., code reuse) by making the address of target code/data unpredictable. However, recent attacks [34, 35] have shown that these mechanisms are not difficult to bypass. As such, more advanced techniques have been proposed. For example, SoftBound [28], CETS [29], and AddressSanitizer [33] (ASan) provide a high memory safety guarantee, CFI [1] and CPI [25] effectively mitigate control flow hijacking attacks, MemorySanitizer [36] (MSan) can mitigate information leaks caused by uninitialized read, and UndefinedBehaviorSanitizer [27] (UBSan) can detect the causes of undefined behaviors (e.g., null pointer dereference).

However, despite the large number of software hardening techniques proposed, few of them actually get adopted in practice. One reason is that the slowdown imposed by these mechanisms erases the performance gains that come from low-level languages. Another reason is that each proposed technique tends to fix only specific issues while leaving the program vulnerable to other attacks. Comprehensive security protection is often demanded by mission-critical services such as web servers or cyber-physical systems in which a single unblocked attack could lead to disastrous consequences (e.g., heartbleed [16]).

In order to achieve comprehensive program protection, an intuitive method is to combine several techniques and enforce them together in a target program. Unfortunately, this is often not viable in practice for two reasons: 1) Runtime slowdown increases unpredictably after fusing different techniques. For instance, in an already highly optimized build [29], combining Softbound and CETS yields a 110% slowdown–almost the sum of each technique individually; 2) Implementation conflicts prevent direct combination because most techniques are not designed with compatibility in mind. For instance, MSan makes the lower protected area inaccessible, while ASan reserves the lower memory as shadow memory. Re-implementing these techniques for better compatibility requires significant engineering effort if it is even possible.

In the meantime, hardware is becoming cheaper and more powerful. The increasing number of CPU cores combined with larger cache and memory size keeps boosting the level of parallelism, making it practical to improve software security through a technique known as N-version programming [5, 9, 20, 44]. As part of this trend, the N-version scheme is particularly suitable to multi-core architectures because replicas can run on cores in parallel.

An N-version system typically requires careful construction of N variants that are both functionally similar in normal situations and behaviorally different when under attacks. Hence, although each program version may be vulnerable to certain types of attacks, the security of the whole system relies on the notion that an attacker has to simultaneously succeed in attacking all variants in order to compromise the whole system. This property of the N-version system gives us insight on how to provide strong security to a program and yet not significantly degrade its end-to-end performance. That is, by distributing the intended security to N program variants and synchronizing their execution in parallel, we can achieve the same level of security with only a portion of its running time plus an overhead for synchronization. Hence, the challenges lie in how to produce the program variants in a principled way and how to synchronize and monitor their executions efficiently and correctly.

In this paper, we introduce BUNSHIN, an N-version-based approach to both minimize the slowdown caused by security mechanisms and seamlessly unify multiple security mechanisms without re-engineering efforts to any individual mechanism. In short, BUNSHIN splits the checks required by security mechanisms and distributes them to multiple variants of a program in automated and principled ways to minimize execution slowdown. By synchronizing the execution of these variants, BUNSHIN guarantees comprehensive security for the target program.

While the N-version mechanism has been well studied primarily for fault-tolerance [5, 9, 21], BUNSHIN aims at improving a program's security and enabling the composition of multiple security mechanisms with automated protection distribution mechanisms. In addition, BUNSHIN is a practical system as it does not require extra modification to the system or compilation toolchain. BUNSHIN supports state-of-the-art mechanisms like ASan, MSan, UBSan, Softbound, and CETS. We have tested it on a number of C/C++ programs, including SPEC2006, SPLASH-2x, PARSEC benchmarks, Nginx, and Lighttpd web servers. Through three case studies, we show that 1) the slowdown for ASan can be reduced from 107% to 65.6% and 47.1% by distributing the sanity checks to two and three variants, respectively; 2) the slowdown for UBSan can be reduced from 228% to 129.5% and 94.5% by distributing the sub-sanitizers to two and three variants, respectively; and 3) the time overhead for unifying ASan, MSan, and UBSan with BUNSHIN is only 4.99% more than the highest overhead of enforcing any of the three sanitizers alone. In summary, our work makes the following contributions:

- We propose an N-version approach to enable different or even conflicting protection techniques to be fused for comprehensive security with efficiency.
- We present an improved NXE design in terms of syscall hooking, multithreading support, and execution optimization.
- We have implemented BUNSHIN and validated the effectiveness of BUNSHIN's NXE and the protection distribution mechanisms in amortizing the slowdown caused by state-of-the-art security mechanisms.

The rest of the paper provides background information and compares BUNSHIN with related works (§2), describes the design and implementation of BUNSHIN (§3, §4), presents evaluation results (§5), discusses its limitations and improvements (§6), and concludes (§7).

2 Background & Related Work

2.1 Memory Errors vs. Sanity Checks

Memory errors occur when the memory is accessed in an unintended manner [37]. The number of reported memory errors is still increasing [14] and severe attacks (e.g., heartbleed [16]) exploiting memory errors emerge from time to time. We provide a taxonomy in Table 1 to summarize the errors. In particular, any vulnerability that may change a pointer unintentionally can be used to achieve out-of-bound reads/writes. Use-after-free and uninitialized read are usually caused by logic bugs (e.g., double-free and use-before-initialization) or compiler features (e.g., padding for alignment). Undefined behaviors can be triggered by various software bugs, such as divide-by-zero and null-pointer dereferences.

Memory Error	Main Causes	Defenses
Out-of-bound r/w	lack of length check format string bug integer overflow bad type casting	SoftBound [28] ASan [33]
Use-after-free	dangling pointer double free	CETS [29] ASan [33]
Uninitialized read	lack of initialization data structure alignment subword copying	MSan [36]
Undef behavior	pointer misalignment divide-by-zero null pointer dereference	UBSan [27]

Table 1: A taxonomy of memory errors. We assume the program is not malware. This taxonomy is mainly derived from two systematic survey papers [37, 38].

How to defend a program against memory errors has been extensively studied in recent years. For each category in Table 1, we are able to find corresponding defenses. In this paper, we are particularly interested in the *sanitizer*-style techniques because they thoroughly enforce sanity checks in the program to immediately catch memory errors before they are exploited.

2.2 N-version System

The concept of the N-version system was initially introduced as a software fault-tolerance technique [9] and was later applied in enhancing software security [5, 20, 39, 44]. In general, the benefit of the N-version system is that an attacker is required to simultaneously compromise all variants with the same input in order to take down the whole system. To achieve this benefit, an N-version system should have at least two components: 1) a variant producer that generates diversified variants based on predefined principles and 2) an execution engine (NXE) that synchronizes and monitors the execution of all program variants. We differentiate BUNSHIN with related works along these two lines of work.

Diversification. Diversification techniques represent the intended protection goal of an N-version system, for example, using complementary scheduling algorithms to survive concurrency errors [39]; using dynamic control-flow diversity and noise injection to thwart cache side-channel attacks [11]; randomizing sensitive data to mitigate privacy leaks [8, 45]; running multiple versions to survive update bugs [20]; using different browser implementations to survive vendor-specific attacks [44]. Diversifica-

tion can also be done in load/run time such as running program variants in disjoint memory layouts to mitigate code reuse attacks [7, 10, 41].

Differently, BUNSHIN aims to reduce the execution slowdown and conflicts of security mechanisms. To achieve this goal, two protection distribution mechanisms are proposed. Partial DA checking [31] also attempted to improve the performance of dynamic analysis with the N-version approach. However, it does not provide any protection distribution mechanism; instead it just insecurely skips checking some syscalls to improve performance. In addition, attacks can be completed by exploiting the vanilla variant before other protected variants find out. In contrast, BUNSHIN proposes two principled diversification techniques to achieve this goal and also presents a more robust NXE with thorough evaluation.

NXE. Depending on how the program variants are generated, an NXE is designed to synchronize variants at different levels, such as instruction/function level [39], syscall level [7, 10, 21, 24, 32, 42] or, file/socket IO level [44]. BUNSHIN shares some common features with other syscall-based NXE systems, including syscall divergence comparison (both sequence and arguments), virtual syscall and signal handling, and a leader-follower execution pattern backed by a ring-buffer-based data structure for efficient event streaming [21, 24, 42]. However, BUNSHIN differs from these works in the following ways:

Syscall hooking. Prior works hook syscalls using a customized kernel [10], which jeopardizes its deployability; using the Linux `ptrace` mechanism, which causes high synchronization overhead due to multiple context switches per each syscall [7, 10, 32, 42]; or binary-rewriting the program to redirect a syscall to a trampoline [21], which may break the semantics of the program when replacing an instruction with fewer bytes with one with more bytes. MvArmor [24] leverages `Dune` [3] and Intel VT-x. However, it incurs a high overhead for syscalls that needs passthrough and is also subject to the limitations of `Dune` (e.g., signal and threading). To tackle these issues, BUNSHIN hooks syscalls by temporarily patching the syscall table with a loadable kernel module.

Multithreading support. Multithreading support varies in proposed NXEs, such as allowing only processes to be forked, not threads [10]; enforcing syscall-ordering across the threads [21, 24, 32], which can easily cause deviations, as not all threading primitives involves syscalls (e.g., `pthread_mutex_lock`); and using CPU page fault exception to synchronize all memory operations [7], which leads to high overhead. None of these works are evaluated on multithreading benchmarks like PARSEC or SPLASH-2x. ReMon [42] seems to have a similar level of support for multithreading as BUNSHIN—race-free programs by injecting synchronization agents into the compiled binary, as discussed in [40]. BUNSHIN borrows the *weak determinism* concept from the deterministic multithreading (DMT) domain and fully describes its design and implementation details to support it. BUNSHIN also identifies its limitations and potential solutions.

2.3 Security/performance Trade-offs

Another approach to fit security into the performance budget is to devise a subset of protections from a full-fledged technique. Compared with Softbound [28], which handles both code and data pointers, CPI [25] only instruments code pointers, which are less prevalent but more critical to code-reuse attacks [34]. Since the number of sanity checks inserted is dramatically reduced, CPI reduces the performance overhead from about 70% to 8.4%. Similarly, ASAP [43] keeps less commonly executed (i.e., less costly) checks and removes hot checks. However, selective protections sacrifice security. The assumption that security is proportional to sanity check coverage is not valid in many cases. More specifically, say a program contains two exploitable buffer overflow vulnerabilities; eliminating only one does not actually improve the security, as one bug is enough for the adversary to launch the attack. Unlike CPI and ASAP, BUNSHIN is a novel concept to reduce the slowdown caused by security mechanisms without sacrificing any security.

3 Design

In a typical N-version system, program variants are executed in parallel and synchronized by an execution engine to detect any behavior divergence. The whole system terminates only when all variants have terminated. Therefore, the overall runtime of an N-version system can be decomposed into two parts: 1) the time required to execute the slowest variant, and 2) any additional time used for variant synchronization and monitoring.

3.1 Protection Distribution Principle

BUNSHIN's protection distribution applies to *sanitizer*-style techniques, which have three properties: 1) They enforce security via instrumenting the program with runtime checks; 2) All the checks instrumented are independent of each other in terms of correctness; and 3) A sanity check alters control flow when and only when the check fails, i.e., the program behaves normally when no memory errors or attacks are present. The majority of memory error prevention techniques are *sanitizer*-style, including stack cookies, CFI, CPI, SAFECode, ASan, MSan, UBSan, Softbound, and CETS, which is also the foundation of profiling-guided security retrofitting such as ASAP [43] and Multicompiler [19].

These properties allow BUNSHIN to 1) measure runtime overhead imposed by the sanitizer as well as remove sanity checks; 2) split the program to allow only portions of the program to be instrumented with sanity checks; 3) split the set of security techniques to allow only selected

checks to be enforced; and 4) produce functionally similar program variants such that BUNSHIN can synchronize their executions and reason about behavior divergences. BUNSHIN distributes checks with two principles.

Check distribution takes a single security technique (e.g., ASan) as it is and distributes its runtime checks on a program over N program variants. Specifically, BUNSHIN first splits the program into several disjoint portions and then generates a set of variants, each with only one portion of the program instrumented by the technique. Since only a fraction of the code is instrumented with security checks, the slowdown for each variant is smaller compared with a fully instrumented program. And given that all portions of the target program are covered through the collection of the variants, the security protection is the same as if the security mechanism is applied to the whole program.

In the example of ASan, the splitting unit is every function in the program and for a 3-variant split, after *check distribution*, each variant has ⅓ functions instrumented with sanity checks and the other ⅔ uninstrumented, while collectively, all functions are covered.

It is worth noting that instrumentations added by sanitizers fall into two categories: metadata maintenance (e.g., bound and alias information in the example of ASan or SoftBound) and sanity checks. BUNSHIN does not remove instructions related to metadata maintenance, as removing them will break the correctness of a sanitizer.

Sanitizer distribution takes multiple security techniques and distributes them over N variants. Specifically, BUNSHIN first splits these security mechanisms into several disjoint groups whereby each group contains security mechanisms that are collectively enforceable to the program, i.e., they do not conflict with each other. Since only a subset of the intended protections is enforced on each variant, the slowdown for each variant is smaller compared with the case when all protection techniques are enforced on the same program (if ever possible). Another important benefit of *sanitizer distribution* is that by distributing security mechanisms to multiple program variants, any conflicts between them (e.g., ASan and MSan) can be avoided without re-engineering these security mechanisms. Since all intended protections are enforced through the collection of variants, the overall protection is the same as if all mechanisms are applied to the whole program.

In the example of UBSan, the splitting unit is every sub-sanitizer in UBSan, such as `integer-overflow` and `divide-by-zero`. For a 3-variant split, after *check distribution*, each variant has a disjoint set of sub-sanitizers (i.e., `integer-overflow` appears in only one variant), while collectively, all sub-sanitizers are covered.

Due to space constraints, interested readers may find a formal modeling of BUNSHIN's protection distribution principles at http://arxiv.org/abs/1705.09165.

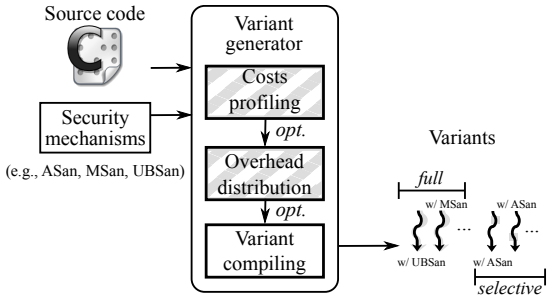


Figure 1: Variant generator workflow

3.2 Automated Variant Generator

Figure 1 illustrates the high-level workflow of the variant generator. The generator first compiles the target program without any security mechanisms enforced and runs it with the profiling tool to get a baseline profile. In the *check distribution* case, the generator then compiles and profiles the program with the intended security mechanism. In the *sanitizer distribution* case, the generator compiles and profiles the program multiple times, each time with one of the intended security mechanisms. The overhead profile is derived by comparing the security-enforced profiles with the baseline profile. In the next step, the generator runs the overhead distribution algorithm with the intended number of splits (the N value) and creates N build configurations for the compilers, each corresponding to one program variant. The goal is to distribute the overhead measured by the profiling step *fairly* to each variants such that all variants finish execution at approximately the same time. Finally, the generator compiles the program N times, each with a different build configuration, to get N program variants.

Profiling. BUNSHIN relies on profiling to obtain the runtime slowdown numbers as the inputs to the overhead distribution algorithm. We choose to explicitly rely on profiling because it is a reliable way to obtain the actual cost of a particular sanity check without making assumptions about the nature of the program or the sanitizer. It also takes in the effect of not only extra CPU cycles required to run the check, but also the side effects on cache-line usage, register pressure, or memory allocations. However, the profiling approach does require an adequate and representative workload to simulate the usage patterns in a production environment. Fortunately, for many projects, such a workload is often available in a form of test suites, which can be directly used to build a profile. More sophisticated profiling tools [2, 15] are orthogonal to BUNSHIN and can be leveraged to improve the overhead profiling if necessary. After profiling, the sanity checks are distributed to N variants in a way that the sum of overhead in each variant is almost the same.

Variant compiling. Variant compiling for *check distribution* is essentially a "de-instrumentation" process that involves deleting the instructions that are only used for

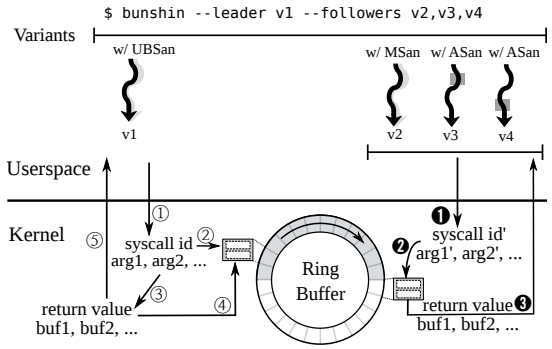


Figure 2: General synchronization procedure. The synchronization is triggered when the syscall is trapped into kernel, as denoted by ①❶ in both the leader and follower path. The leader then checks-in the syscall arguments to the shared slot (②), executes the syscall (③), and turns-in the execution results in the shared slot (④). A follower first checks whether the syscall arguments stored in the slot match its own arguments (❷) and if they match, directly fetches the results from sync slot (❸) without actually performing the syscall. The difference between *lockstep* mode and *ring-buffer* mode lies in whether step ③ for the leader can be performed before step ❷ for all followers is completed.

sanity checks instrumented by sanitizers. In order to collect such instructions for deletion, BUNSHIN uses data and control dependence information maintained during the compilation process and performs *backward slicing* to automatically collect sanity check-related instructions and discard them. Variant compiling for *sanitizer distribution* is trivial, as it can be done by simply compiling the program with the compilation settings a user would normally use for those sanitizers, as long as the sanitizers used to harden the program are collectively enforceable.

3.3 N-version Execution Engine

BUNSHIN's NXE synchronizes the executions of N program variants and makes them appear as a single instance to any external entity. We present and justify various design efforts to improve BUNSHIN's NXE in efficiency and robustness.

Strict- and selective- lockstep. To synchronize the leader and the follower instances, a lockstep at syscalls is required. BUNSHIN provides two lockstep modes: *strict-lockstep* and *selective-lockstep*. In *strict-lockstep* mode, the leader executes the syscall only if all followers have arrived and agreed on the syscall sequence and arguments. This ensures the security guarantee—the attack cannot complete in either instance. The downside is that variants are frequently scheduled in and out of the CPU due to the necessary waiting, leading to higher runtime slowdown.

We observed that many attacks always trigger certain syscalls before the actual damages are caused. For example, with ASLR enabled, attacks (e.g., ROP) generally leak an address first via I/O write syscalls and then use the leaked address to construct subsequent attack payloads. Based on this observation, BUNSHIN also provides the *selective-lockstep* so that users can choose to prevent the attacks with higher performance. Specifically, BUNSHIN uses the *ring-buffer* mechanism to synchronize instances—the leader executes at near full speed and keeps dumping the syscall arguments and results into the shared ring buffer without waiting, unless the buffer is full. The followers consume the syscall arguments and results at their own speed. Meanwhile, lockstep is still enforced for the selected syscalls (e.g., write related), as illustrated in Figure 2. Our evaluation §5.2, shows that *selective-lockstep* reduces the synchronization overhead by 0.3%-6.3% compared with the *strict-lockstep* mode.

In short, *strict-lockstep* should perfectly preserve the security guarantee of the underlying sanitizer, while *selective-lockstep* is an option we provide when ASLR is enabled, as any remote code-reuse attacks (e.g., return-to-libc and ROP) will have to first leak code/data pointers to bypass ASLR. *Selective-lockstep* is able to stop such attacks by catching the leaks at I/O writes. In fact, any information leak attempt that involves a pointer will be detected at I/O writes. A detailed analysis of the security guarantees provided by BUNSHIN is evaluated in §5.3.

Multi-threading. BUNSHIN supports multi-process/thread programs by assigning each group of leader-follower processes to the same *execution group*, and each execution group has its own shared buffers. The starting processes of leader and follower variants form the first execution group, and when the leader forks a child, the child automatically becomes the leader in the new execution group. The child of a follower variant automatically becomes a follower in the new execution group. In fact, for daemon-like programs, (e.g., Apache, Nginx, sshd), simply separating parent and children processes into different execution groups can be sufficient to eliminate syscall sequence variations caused by non-deterministic schedulers because for those programs, each thread/process is highly independent of the others and hardly ever or never updates shared data.

However, for general-purpose multi-thread programs, synchronizing shared memory accesses is necessary to ensure that the leader and followers have consistent views on shared data. This can be achieved by enforcing all followers to follow exactly the same order of shared memory accesses as the leader (*strong determinism*), which can hardly be achieved without a high performance penalty, as evidenced in the deterministic multi-threading (DMT) domain [4, 30]. As a compromise, inspired by Kendo [30], BUNSHIN ensures only that all followers follow exactly the same order of all lock acquisitions as the leader (*weak determinism*). For example, if thread 1 in the leader acquires a mutex before thread 2 passes a barrier, the same order will be enforced in all followers. For programs without data races, strong determinism and weak determinism

offer equivalent guarantees [30]. BUNSHIN achieves this with an additional 8.5% overhead on SPLASH-2x and PARSEC benchmarks (§5.2).

We argue that ensuring *weak determinism* is sufficient for the majority of multi-thread programs, as race-free programming is encouraged and tools have been proposed to help developers eliminate data races [17, 18]. However, should this becomes a problem in the future, BUNSHIN is capable of plugging in sophisticated DMT solutions such as DThreads [26] with minor adjustments.

Sanitizer-introduced syscalls. Memory safety techniques generally issue additional syscalls during program execution to facilitate sanity checks. With all sanitizers we tested, i.e., ASan, MSan, UBSan, Softbound, CETS, CPI, and SAFECode, all introduced syscalls can be categorized into three classes: 1) pre-launch data collection, 2) in-execution memory management, and 3) post-exit report generation. To illustrate, before executing the `main` function, ASan goes through a data collection phase by reading various files in `/proc/self` directory (on Linux system). During program execution, ASan issues additional memory-related syscalls for metadata management. Upon program exit, ASan might invoke external programs to generate human readable reports.

Given that variants instrumented with different sanity checks are expected to have diverged syscall sequences, BUNSHIN needs to address this issue to avoid false alerts. In achieving this, BUNSHIN 1) starts synchronization only when a program enters its `main` function; 2) ignores all the memory management-related syscalls; and 3) stops synchronization by registering as the first `exit` handler.

We verified that all the syscall divergences caused by the aforementioned sanitizers are successfully resolved. Although this is only an empirical verification, we believe this can be a general solution because any practical security mechanism should not alter program semantics in normal execution states, which, reflected to the outside entities, are syscall sequences and arguments.

4 Implementation

4.1 Automated Variant Generator

Profiling. To obtain overhead data for *check distribution*, BUNSHIN instruments the program with performance counters based on how the underlying sanitizer works. As a prototype system, BUNSHIN currently measures the execution time of all program functions based on the observation that the majority of memory-related security checks (as discussed in §2.1) operate at function level. We discuss how to perform profiling instrumentation in a generic way in §6. Obtaining profiling data for *sanitizer distribution* is easy, as no extra instrumentation is needed. BUNSHIN runs the program with each security mechanism individually enforced and obtains the overall execution time.

Check removal. BUNSHIN removes sanity checks at function level and the process consists of two steps:

In the *discovery* step, BUNSHIN compiles a baseline version and an instrumented version of the same program and then uses an analysis pass to dump the added/modified basic blocks per function. Among these basic blocks, BUNSHIN considers a basic block that 1) is a branch target, 2) contains one of the known sanity check handler functions (e.g., in the case of ASan, functions prefixed with `__asan_report_`), and 3) ends with the special LLVM `unreachable` instruction as a *sink* point for security checks. This is based on the properties of sanitizers, as a sanity check should preserve program semantics, i.e., special procedures are only invoked when a sanity check fails. Instrumentations for metadata maintenance involves neither sanity check functions nor `unreachable` instructions and hence are filtered out in this step.

In the *removal* step, BUNSHIN automatically reconstructs sanity checks based on the observation that sanity checks are instructions that branch to the *sink* points found in *discovery* step. After identifying the branching points and the corresponding condition variables, BUNSHIN performs a recursive backward trace to variables and instructions that lead to the derivation of the condition variable and marks these instructions during tracing. The backward trace stops when it encounters a variable that is not only used in deriving the value of the condition variable but also used elsewhere in the program, an indication that it does not belong to the sanity check. Removing the sanity check is achieved by removing all marked instructions found in the above process. This functionality is implemented as an LLVM pass.

4.2 N-version Execution Engine

Pthreads locking primitives. BUNSHIN enforces *weak determinism* discussed in §3.3 by re-implementing the full range of synchronization operations supported by `pthreads` API, including locks, condition variables, and barriers. BUNSHIN introduces a new syscall, `synccall`, specifically for this purpose. `synccall` is exposed to processes under synchronization by hooking an unimplemented syscall in an x86-64 Linux kernel (`tuxcall`). In the kernel module, BUNSHIN maintains an `order_list` to record the total ordering of locking primitive executions. When a leader thread hits a primitive, e.g., `pthread_mutex_lock`, it calls `synccall` to atomically put its *execution group* id (EGID) in the `order_list` and wake up any follower threads waiting on its EGID before executing the primitive. When a follower thread hits a primitive, the call to `synccall` will first check whether it is the thread's turn to proceed by comparing its EGID and the next EGID in the `order_list`. The thread will proceed if it matches; otherwise, it puts itself into a variant-specific waitqueue. If the primitive may cause the thread to sleep

(such as a mutex), the thread wakes up its next sibling in the waitqueue, if there are any, before sleeping.

Hooking `pthreads`' locking primitives is done by placing the patched primitives in a shared library, which is guaranteed to be loaded earlier than `libpthread`.

The drawbacks of this implementation are also obvious: 1) BUNSHIN is unable to handle multi-thread programs that are built with other threading libraries or that use non-standard synchronization primitives (e.g., using `futex` directly); 2) The performance overhead of BUNSHIN increases linearly with the usage frequency of these primitive operations. Fortunately, the majority of multi-thread programs are compatible with `pthreads` and locking primitives are only used to guard critical sections, which represent only a small fraction of execution.

Shared memory access. Similar to the approach used by MSan to trace uninitialized memory accesses, whenever BUNSHIN detects mapping of shared memory into the variant's address space (by the indication of `mmap` syscall with specific flag combinations), it creates shadow memory copies of the same size and then marks them as "poisoned" state `HWPOISON` whereby any access attempts to the mapped memory will also lead to an access attempt to the shadow copy, which eventually triggers a signal (`SIGBUS`). Upon capturing the signal, memory access is synchronized in the normal way syscalls are handled, i.e., compare and copy content of the accessed address from the leader's mapping to the followers mapping.

Workflow. BUNSHIN can be started with the path to each variant and the program arguments. BUNSHIN first informs the kernel module to patch the syscall table and then sets the `LD_PRELOAD` environment variable to the library containing patched virtual syscalls and pthread locking primitives. It then forks N times and launches one program variant in each child process. After that, BUNSHIN pauses and waits for status change of the variants. If any of the variant process is killed due to behavior divergences, BUNSHIN alerts and aborts all variants. Otherwise, it exits when all variants terminate.

5 Evaluation

In this section, we first evaluate BUNSHIN's NXE in terms of robustness and efficiency. In particular, we run BUNSHIN on various programs and empirically show that BUNSHIN is capable of handling the majority of them with low overhead and no false alarms. We also empirically test whether BUNSHIN can provide the same level of security guarantee as the underlying sanitizers, in other words, whether BUNSHIN might compromise the security by partitioning the program or splitting the sanitizers.

We then showcase how to accelerate ASan-hardened programs with *check distribution* and UBSan-hardened programs with *sanitizer distribution*. We use another case study – combining ASan, MSan, and UBSan – to show that BUNSHIN is capable of unifying security mechanisms that have conflicted implementations.

We also evaluate BUNSHIN in terms of hardware resource consumption, which could limit BUNSHIN's applicability, and report the performance of BUNSHIN under various levels of system load.

Experiment setup. The experiments are primarily conducted on a machine with Intel Xeon E5-1620 CPU (4 cores) and 64GB RAM running 64-bit Ubuntu 14.04 LTS, except the experiment on scalability, which is done with Intel Xeon E5-2658 (12 cores), and the experiment on RIPE benchmark, which is done on a 32-bit virtual machine. For evaluations on web servers, we dedicate another machine to launch requests and measure server response time. The client machine is connected to the experiment machine with a direct network cable. The associated network card permits 1000Mb/s bandwidth. Unless stated otherwise, the NXE is configured to run in *strict-lockstep* mode for stronger security guarantee.

5.1 NXE Robustness

We use a mixed sample of CPU-intensive and IO-intensive programs for experiments, including SPEC2006 benchmark representing single-thread programs, PARSEC, and SPLASH-2x benchmark for multi-threaded programs, and Nginx and Lighttpd as representative server programs. For each sample program, we compile it with the LLVM compiler framework and run the same binary on BUNSHIN's NXE, i.e., BUNSHIN will synchronize identical N binaries. This is to (empirically) verify the robustness of BUNSHIN's NXE design.

We do not observe false positives in any experiments on SPEC, SPLASH-2x, Nginx, and Lighttpd. However, BUNSHIN is only able to run on six out of 13 programs in the PARSEC benchmark. `raytrace` would not build under `clang` with `-flto` enabled. `canneal`, `facesim`, `ferret`, and `x264` intentionally allow for data races. `fluidanimate` uses ad-hoc synchronization and hence, bypassing `pthreads` APIs and `freqmine` does not use `pthreads` for threading. These represent the limitation of BUNSHIN's NXE: enforcing only weak determinism on `pthreads` APIs.

5.2 NXE Efficiency & Scalability

Figure 3 and Figure 4 show the efficiency evaluation of the NXE under both strict- and selective-lockstep modes when synchronizing 3 variants. For the SPEC2006 benchmark, the average slowdowns for the two modes are 8.1% and 5.3%, respectively. The overhead is sightly higher on multi-threaded benchmarks (SPLASH-2x and PARSEC) – 15.7% and 13.8%. This is due to the additional overhead for recording and enforcing the total order of locking primitive acquisitions. The selective-lockstep mode reduces the overhead by 0.3%-6.3% in the benchmark programs.

We further evaluate the efficiency of BUNSHIN's NXE

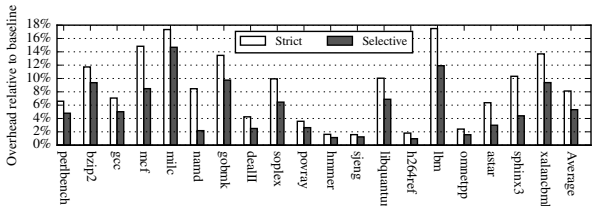


Figure 3: Evaluation of BUNSHIN' NXE efficiency with SPEC2006.

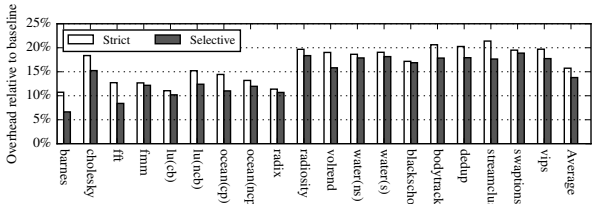


Figure 4: Evaluation of BUNSHIN's NXE efficiency with SPLASH-2x and PARSEC (number of threads = 4).

on two server programs, `lighttpd`, representing single-thread servers, and `nginx`, representing multi-threaded servers. We synchronize 3 variants and for `nginx`, we run 4 worker threads, the default value after installation. We simulate various workload situations by using 64 (light), 512 (heavy), and 1024 (saturated) concurrent connections and simulate HTTP requests to files of 1KB and 1MB.

The results are shown in Table 2. A noticeable difference is that the percentage overhead when requesting small files (e.g., 1KB), is significantly larger compared with requesting large files (e.g., 1MB). The reason is that, while the absolute value of overhead is comparable in both situations, it can be better amortized into the networking time of a large file, therefore leading to smaller relative overhead. We believe that in real-world settings when the servers are connected to LANs and WANs, even the overhead for smaller files can be amortized in the networking time, leading to unnoticeable overhead.

Figure 5 shows the scalability of BUNSHIN's NXE in terms of total number of variants synchronized. We use a 12-core machine for this experiment as the number of variants should not exceed the number of cores available. As the number of variants goes from 2 to 8, the overhead increases from 0.9% to 21% accordingly. The primary reason for overhead increase is the LLC cache pressure, as the cache miss rates increase exponentially when more variants are executed in parallel. Recently added CPU features such as Intel Cache Allocation Technology [22] might help to mitigate this problem.

5.3 Security Guarantee

BUNSHIN does not remove any sanity checks, but only distributes them into multiple variants. In *strict-lockstep* mode, BUNSHIN should not compromise the intended security guarantee, as no variant can proceed with a syscall without the arrival of other variants. Conceptually, the only way to compromise all variants is to launch an attack that is out of the protection scope of the underlying

Config	Conn	Base	Strict		Selective	
`lighttpd`	64	10.3	11.9	15.3%	11.8	14.6%
1 Process	512	8.71	10.5	20.5%	10.1	15.7%
1K File	1024	8.48	10.4	22.6%	10.1	19.3%
`lighttpd`	64	974	994	2.05%	992	1.85%
1 Process	512	959	972	1.35%	970	1.15%
1M File	1024	955	964	0.91%	961	0.63%
`nginx`	64	9.81	11.6	18.7%	11.2	14.3%
4 Threads	512	8.46	10.3	21.9%	9.88	16.8%
1K File	1024	8.20	10.2	24.4%	9.63	17.4%
`nginx`	64	950	967	1.79%	964	1.47%
4 Threads	512	985	999	1.40%	996	1.12%
1M File	1024	979	998	1.94%	995	1.63%
Ave. (1KB)				20.56%		16.4%
Ave. (1MB)				1.57%		1.31%

Table 2: Performance of `lighttpd` and `nginx` under BUNSHIN's NXE, Performance measured as the processing time per request (unit. μs). We use `apachebench` as test driver and run each experiment 1000 times to reduce the effect of network noise.

sanitizer. Given that there is no attack window between variants, if an input sequentially compromises all variants without causing a divergence, it means the input bypasses all sanity checks. In this case, the attack will also succeed even if all checks are enforced in one variant (i.e., no BUNSHIN). In other words, the attack is essentially not in the protection scope of the underlying sanitizers and hence will not be in the scope of BUNSHIN.

On the other hand, the *selective-lockstep* mode might introduce an attack window between the variants that allow an attacker to potentially compromise them one by one. However, BUNSHIN remains effective if the window is small enough. To quantify the attack window, we measure the syscall distance between the leader and the slowest follower during our experiments. For CPU-intensive programs (SPEC2006, PARSEC, and SPLASH-2x), the average number of syscall gap is 5 while for IO-intensive programs (lighttpd and nginx), the average number of syscall gap is only 1. The gap is small because even in *selective-lockstep* mode, the variants are still strictly synchronized at IO-related syscalls. We believe that this is a small enough time frame to thwart attackers.

To empirically confirm that real-world attacks can be thwarted even in *selective-lockstep* mode, we first evaluated BUNSHIN on the RIPE benchmark with *check distribution* on ASan. In particular, in compiling the programs generated by the RIPE benchmark, we go through the normal *check distribution* procedure to produce two variants, and then launch and synchronize them with our NXE. The results in Table 3 confirm that BUNSHIN does not compromise the intended security guarantee of ASan.

To further verify this, we applied BUNSHIN to five real-world programs, `nginx`, `cpython`, `php`, `openssl`, and `httpd`, which contain known vulnerabilities that can be detected by ASan (to evaluate *check distribution*) and UBSan (to evaluate *sanitizer distribution*). Similar to the RIPE benchmark case, we apply BUNSHIN on these vulnerable programs to produce two variants and later

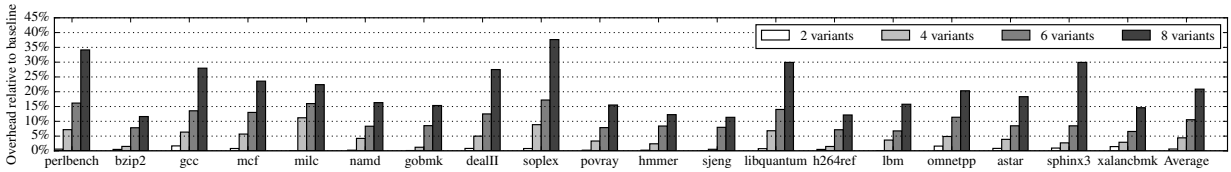


Figure 5: Scalability of BUNSHIN in terms of synchronizing 2 to 8 variants. For each program, we show the synchronization overhead over the baseline execution. On average, the overhead almost doubled with 2 more variants synchronized. Different programs show slightly different patterns in overhead growth. One of the reasons could be their differences in cache sensitivity [23].

Config	Succeed	Probabilistic	Failed	Not possible
Default	114	16	720	2990
ASan	8	0	842	2990
BUNSHIN	8	0	842	2990

Table 3: We first run the RIPE benchmark on vanilla 32-bit Ubuntu 14.04 OS, and 114 exploits always succeed and 16 succeed probabilistically. After adding ASan in the compilation, only 8 exploits succeed. After applying *check distribution* on the programs, still the same 8 exploits succeed.

Program	CVE	Exploits	Sanitizer	Detect
nginx-1.4.0	2013-2028	blind ROP	ASan	Yes
cpython-2.7.10	2016-5636	int. overflow	ASan	Yes
php-5.6.6	2015-4602	type confusion	ASan	Yes
openssl-1.0.1a	2014-0160	heartbleed	ASan	Yes
httpd-2.4.10	2014-3581	null deref.	UBSan	Yes

Table 4: Empirically test BUNSHIN's security guarantee with real-world programs and CVEs.

launch and synchronize them with our NXE. We then use the same exploit that triggered the warnings from ASan or UBSan to drive the program under BUNSHIN and check to see whether the same warnings are raised. The result show that all exploitation attempts are detected (Table 4).

For a concrete example, we applied BUNSHIN to `nginx-1.4.0`, which contains bug CVE-2013-2028 that can be detected by ASan. We use *check distribution* to produce two variants, A and B and use three published exploits [6, 12, 13] to test whether they can succeed in exploiting the vulnerable `nginx` protected under BUNSHIN. The result shows that, when the overflow is triggered, variant A issues a `write` syscall (trying to write to `stderr`) due to ASan's reporting, while B does not. A further investigation on the protection distribution report shows that the vulnerable function `ngx_http_parse_chunked` is instructed to be protected by variant A, which explains why variant A issues the `write` syscall.

Attacking BUNSHIN. Given the attack window in *selective-lockstep* mode, an attacker might be able to complete some simple attacks before detection, such as killing child threads/processes, closing file/sockets, or exhausting resources by allocating large chunks of memory, etc., provided that the attacker can inject shellcode or reuse program code to invoke the call and place the arguments of syscall in correct addresses. An attacker might also launch denial-of-service attacks by sending compromised variants into infinite loops that do not involve synchronize syscalls (in both modes) or sleep/pause indefinitely (in *selective-lockstep* mode).

Another attack vector is BUNSHIN's variant monitor. For example, an attacker might intentionally crash BUNSHIN with unhandled non-deterministic sources such as uninitialized data (e.g., some encryption libraries intentionally use uninitialized data as a source of entropy, although such a practice is discouraged). In addition, although we take care to keep the variant monitor simple and secure, it is not guaranteed to be bug-free. Therefore, if an attacker compromises the variant monitor, he/she might be able to circumvent syscall synchronization.

5.4 *Check Distribution* on ASan

We show the effectiveness of *check distribution* in accelerating the performance of programs instrumented with ASan. The reason we choose ASan for the case study is twofold: 1) ASan is representative of how memory error detection techniques are generally enforced – introducing runtime sanity checks. In addition, the majority of checks placed in the program are independent of each other and hence satisfy the assumption of *check distribution*. 2) ASan provides a relatively high coverage on memory safety and hence is appealing for long-living processes (like server programs) to thwart attackers at runtime. However, the slowdown by enforcing ASan to the whole program is the main obstacle in making it useful in production. We hope this experiment will provide insights on how to use ASan through BUNSHIN.

The case study is done with the SPEC2006 benchmark programs using the `train` dataset for profiling and `reference` dataset for the actual performance measurement. On average, the runtime slowdown caused by ASan is reduced from 107% (enforced to the whole program) to 65.6% (2 variants) and 47.1% (3 variants), respectively, about 11% more than ½ and ⅓ of the original slowdown. Due to space constraints, we show only results for the more complex case (3 variants) in Figure 6.

However, we also observed two outliers that do not show overhead distribution: `hmmer` and `lbm`. After investigating their execution profile, we observe that there is a single function that accounts for over 95% of the execution time and the slowdown caused by ASan. Since BUNSHIN performs sanity check distribution at the function level, the overhead is inevitably distributed to one variant, causing that variant to be the bottleneck of the entire system. However, concentrating functionalities in one

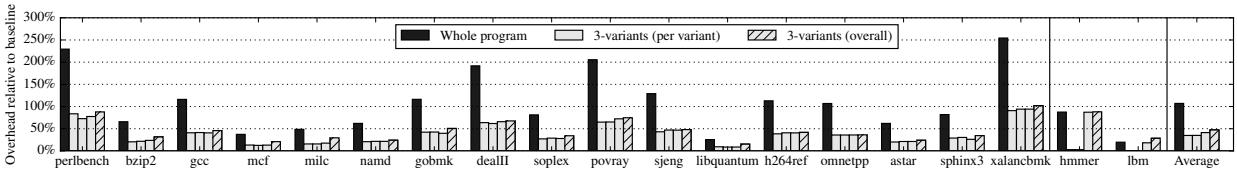


Figure 6: Effectiveness of *check distribution* on ASan with three variants. For each program, we show the total overhead if ASan is applied to the whole program as well as per-variant overhead and BUNSHIN overall overhead. The two programs on the right are outliers that do not show overhead distribution.

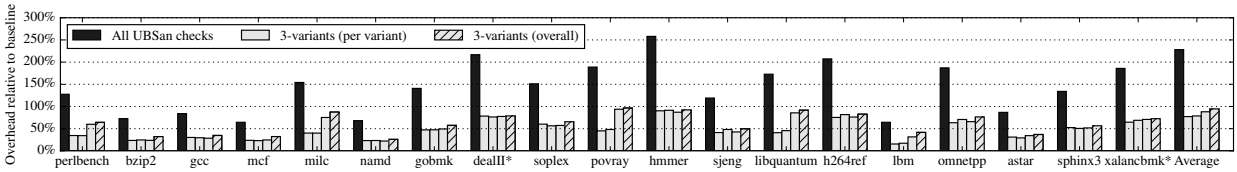


Figure 7: Effectiveness of *sanitizer distribution* on UBSan with three variants. For each program, we show the total overhead if all checks of UBSan are enforced as well as per-variant overhead and BUNSHIN overall overhead. For `dealII` and `xalancbmk`, the overhead number is 4x larger than what is shown in the figure.

single function is rarely seen in the real-world software we tested, including Python, Perl, PHP Apache httpd, OrzHttpd, and OpenSSL; hence, we do not believe these outliers impair the practicality of BUNSHIN.

5.5 *Sanitizer Distribution* on UBSan

UBSan is a representative example to illustrate why collectively enforcing lightweight sanity checks might lead to significant overall slowdown. UBSan contains 19 sub-sanitizers, each with overhead no more than 40%. However, adding them leads to over 228% overhead on SPEC2006 benchmarks, making UBSan a perfect example to exercise *sanitizer distribution*.

Similar to the ASan case study, the test case is done with SPEC2006 programs using `train` dataset for profiling and `reference` dataset for experimentation. On average, the runtime caused by UBSan is reduced from 228% (enforced all checks) to 129% (2 variants) and 94.5% (3 variants), respectively, about 15% more than ½ and ⅓ of the original slowdown. Due to space constraints, we show only the results for the more complex case (3 variants) in Figure 7. This deviation from the theoretical optimum is a bit larger compared with the ASan case study because we only have 19 elements in the set and hence are less likely to get balanced partitions across variants. However, it still shows the effectiveness of *sanitizer distribution* in accelerating the overall performance.

5.6 Unifying LLVM Sanitizers

In theory, BUNSHIN is capable of unifying any security mechanism that falls in the *sanitizers* definition in §3.1. The reason we choose LLVM sanitizers (ASan, MSan, UBSan) for the case study is mainly because: collectively, they provide almost full protection against memory error, which we have not seen in any other work. Unifying them through BUNSHIN might give some insight on how to achieve full memory error protection without any re-engineering effort to these sanitizers.

In this case study, each variant is simply the program compiled with one of the sanitizers with the default compilation settings. We measure the execution time of each program variant when running by itself and also the total execution time of BUNSHIN. The result is reported in Figure 8. On average, the total slowdown of combining these sanitizers is 278%, with only 4.99% more compared with merely enforcing the slowest sanitizer among the three. In other words, paying a little slowdown for synchronization helps bring additional protection provided by the other two sanitizers.

5.7 Hardware Resource Consumption

Memory. Since all variants are loaded into memory for parallel execution, the basic memory usage is almost linear to the number of variants. This is an inherent trade-off for execution time. In addition, whether *check distribution* helps to split memory overhead caused by a sanitizer depends on the sanitizer' internal working, In the case study of ASan, although each variant executes only a portion of the sanity checks, it still needs to shadow the whole memory space as required by ASan. Therefore, the memory overhead of ASan still applies to each variant. However, the memory overhead can be distributed for shadow stack-based techniques. By definition, *sanitizer distribution* can be used to distribute memory overhead to multiple variants. In the UBSan case study, the memory overhead of each variant is the sum of all enforced sub-sanitizers' overhead.

CPU cycles. BUNSHIN's NXE utilizes spare cycles in a multi-core CPU for efficient variant synchronization. If the CPU does not provide sufficient parallelism, BUNSHIN will not be able to improve the performance; instead, it will only introduce more performance overhead. An evaluation on C/C++ programs in the SPEC2006 benchmark shows that the average synchronization overhead is 103.1% when running BUNSHIN on a single core.

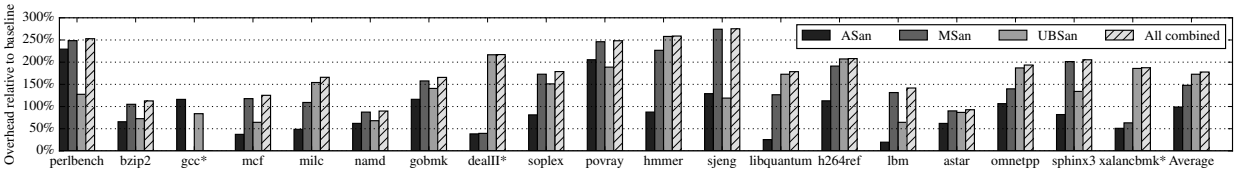


Figure 8: Performance result of each LLVM sanitizer respectively as well as the overall performance overhead when unified under BUNSHIN. `gcc` cannot run with MSan, therefore, we exclude the evaluation on it. For `dealII` and `xalancbmk` the overhead number is 4x larger than what is shown in the figure.

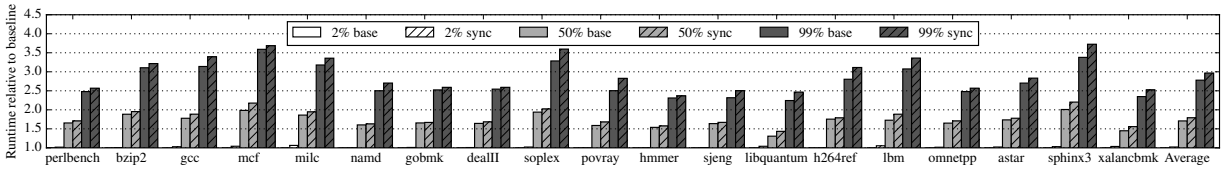


Figure 9: Evaluation of BUNSHIN execution engine under various workload levels. The experiment is done with the configuration of synchronizing 2 variants. We use the system-stressing tool `stress-ng` to add background workloads including CPU tasks, cache thrashing, and memory allocations and deallocations. We maintain the background load level at 50% and 99%, respectively. The 2% load for the baseline case is due to the kernel and OS background services.

Although BUNSHIN is not suitable for devices with a single core, it does not mean that BUNSHIN requires exclusive cores to work. In fact, due to OS-level task scheduling, BUNSHIN can exploit free cycles in the CPU as long as not all cores are fully utilized. Figure 9 shows that BUNSHIN's performance is stable under various load levels. The average slowdown due to synchronization is 10.23% and 13.46%, respectively, when the CPU is half and fully loaded, slightly higher than the case when the load is small (8.1%). The results prove that the performance of BUNSHIN is stable across various load levels.

6 Discussion

Trading-off resources for time. There is no doubt that BUNSHIN's parallelism consumes more hardware resources; hence BUNSHIN is not suitable for cases where hardware resources are scarce. In fact, BUNSHIN's design is inspired by the popularity of multi-core processors and large-size cache and memory, and trade-off resource usages for execution time. BUNSHIN empowers users to make use of available hardware resources to improve both security and runtime performance and sheds lights on how to solve a difficult problem —speeding up hardened programs without sacrificing security —with simply more hardware resources, which are easy to obtain.

Sanitizer integration. BUNSHIN currently has no integration with the sanitizers, i.e., it does not require detailed knowledge of how a sanitizer works in order to "de-instrument" the sanity checks. Although this gives BUNSHIN great flexibility, it also prevents BUNSHIN from further optimization. For example, ASan still shadows the whole memory space even when only a subset of sanity checks is performed per program variant, thus leading to increased memory usage in every variant. To solve this, we could modify ASan's logic in a way such that only a portion of the memory space is shadowed in each variant; in other words, we can distribute the memory overhead to all program variants as well.

Finer-grained sanity check distribution. As shown in the case of `hmmer` and `lbm`, sanity check distribution at the function level might be too coarse grained if one or a few functions dominate the total execution. Therefore, to enable finer-grained overhead distribution, we plan to look into performing both profiling and check removal at the basic block level.

7 Conclusion

We presented BUNSHIN, an N-version system that seamlessly unifies different and even conflicting protection techniques while at same time reducing execution slowdown. BUNSHIN achieves this with two automated variant generation strategies (*check distribution* and *sanitizer distribution*) for distributing security checks to variants and an efficient parallel execution engine that synchronizes and monitors the behaviors of these variants. Our experiment results show that BUNSHIN is a practical system that can significantly reduce slowdown of sanitizers (e.g., 107% to 47.1% for ASan, 228% to 94.5% for UBSan) and collectively enforce ASan, MSan, and UBSan without conflicts with only 4.99% incremental overhead.

8 Acknowledgment

We thank our shepherd, Ittay Eyal, and the anonymous reviewers for their helpful feedback. This research was supported by NSF under award DGE-1500084, CNS-1563848, CRI-1629851, CNS-1017265, CNS-0831300, and CNS-1149051, ONR under grant N000140911042 and N000141512162, DHS under contract No. N66001-12-C-0133, United States Air Force under contract No. FA8650-10-C-7025, DARPA under contract No. DARPA FA8650-15-C-7556, and DARPA HR0011-16-C-0059, and ETRI MSIP/IITP[B0101-15-0644].

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, November 2005.

[2] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Snowbird, Utah, June 2001.

[3] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazieres, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, October 2012.

[4] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: A compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Pittsburgh, PA, March 2010.

[5] Philippe Bergheaud, Dinesh Subhraveti, and Marc Vertes. Fault tolerance in multiprocessor systems via application cloning. In *Proceedings of the 27th IEEE International Conference on Distributed Computing Systems (ICDCS)*, Toronto, Canada, June 2007.

[6] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. Blind return oriented programming (brop), 2014. http://www.scs.stanford.edu/brop.

[7] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. DiversifiedÃ process replicÃę for defeating memory error exploits. In *Proceedings of the 2007 International Performance, Computing, and Communications Conference (IPCCC)*, New Orleans, LA, April 2007.

[8] Roberto Capizzi, Antonio Longo, V. N. Venkatakrishnan, and A. Prasad Sistla. Preventing information leaks through shadow executions. In *Proceedings of the 2008 International Conference on Software Engineering (ICSE)*, Anaheim, CA, December 2008.

[9] Liming Chen and Algirdas Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Fault-Tolerant Computing, 1995*, Jun. 1995.

[10] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: A secretless framework for security through diversity. In *Proceedings of the 15th Usenix Security Symposium (Security)*, Vancouver, Canada, July 2006.

[11] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting cache side-channel attacks through dynamic software diversity. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[12] Exploit Database. nginx 1.3.9-1.4.0 - dos poc, 2013. https://www.exploit-db.com/exploits/25499.

[13] Exploit Database. nginx 1.3.9/1.4.0 x86 - brute force exploit, 2013. https://www.exploit-db.com/exploits/26737.

[14] CVE Details. Vulnerabilities By Date, 2016. http://www.cvedetails.com/browse-by-date.php.

[15] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: less is more. *ACM SIGARCH Computer Architecture News*, December 2000.

[16] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, 2014.

[17] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. Ifrit: interference-free regions for dynamic data-race detection. In *Proceedings of the 23th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Tucson, AZ, October 2012.

[18] Cormac Flanagan and Stephen N Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (PoPL)*, Venice, Italy, January 2004.

[19] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. Profile-guided Automated Software Diversity. In *Proceedings of the 2013 International Symposium on Code Generation and Optimization (CGO)*, Shenzhen, China, February 2013.

[20] Petr Hosek and Cristian Cadar. Safe software updates via multi-version execution. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, May 2013.

[21] Petr Hosek and Cristian Cadar. Varan the unbelievable, an efficient n-version execution framework. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Istanbul, Turkey, March 2015.

[22] Intel. Improving real-time performance by utilizing cache allocation technology, 2015. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf.

[23] Aamer Jaleel. Memory characterization of workloads using instrumentation-driven simulation, 2010. http://www.jaleels.org/ajaleel/publications/SPECanalysis.pdf.

[24] Koen Koning, Herbert Bos, and Cristiano Giuffrida. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *Proceedings of the 46th International Conference on Dependable Systems and Networks (DSN)*, Toulouse, France, June 2016.

[25] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Down Song. Code pointer integrity. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, October 2014.

[26] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient and deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, October 2011.

[27] LLVM. UndefinedBehaviorSanitizer (UBSan) is a fast undefined behavior detector, Feb. 2015. http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[28] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Dublin, Ireland, June 2009.

[29] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: Compiler enforced temporal safety for C. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM)*, Toronto, Canada, June 2010.

[30] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Washington, DC, March 2009.

[31] Luis Pina and Cristian Cadar. Towards deployment-time dynamic analysis of server applications. In *Proceedings of the 13th Inter-*

national Workshop on Dynamic Analysis (WODA), Pittsburgh, PA, October 2015.

[32] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the ACM EuroSys Conference*, Nuremberg, Germany, March 2009.

[33] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 ATC Annual Technical Conference (ATC)*, Boston, MA, June 2012.

[34] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, October–November 2007.

[35] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, October 2004.

[36] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the 2015 International Symposium on Code Generation and Optimization (CGO)*, San Francisco, CA, February 2015.

[37] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[38] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors: The past, the present, and the future. In *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, Amsterdam, Netherlands, September 2012.

[39] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Detecting and surviving data races using complementary schedules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, October 2011.

[40] Stijn Volckaert, Bart Coppens, Bjorn De Sutter, Koen De Bosschere, Per Larsen, and Michael Franz. Taming Parallelism in a Multi-Variant Execution Environment. In *Proceedings of the ACM EuroSys Conference*, Belgrade, Serbia, April 2017.

[41] Stijn Volckaert, Bart Coppens, and Bjorn De Sutte. Cloning your gadgets: Complete ROP attack immunity with multi-variant execution. *IEEE Transactions on Dependable and Secure Computing*, 13(4):437–450, July 2016.

[42] Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. Secure and efficient application monitoring and replication. In *Proceedings of the 2016 ATC Annual Technical Conference (ATC)*, Denver, CO, June 2016.

[43] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High system-code security with low overhead. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[44] Hui Xue, Nathan Dautenhahn, and Samuel T. King. Using replicated execution for a more secure and reliable web browser. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2012.

[45] Aydan Yumerefendi, Benjamin Mickle, and Landon P. Cox. Tightlip: Keeping applications from spilling the beans. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, Cambridge, MA, April 2007.

Glamdring: Automatic Application Partitioning for Intel SGX

Joshua Lind
Imperial College London

Christian Priebe
Imperial College London

Divya Muthukumaran
Imperial College London

Dan O'Keeffe
Imperial College London

Pierre-Louis Aublin
Imperial College London

Florian Kelbert
Imperial College London

Tobias Reiher
TU Dresden

David Goltzsche
TU Braunschweig

David Eyers
University of Otago

Rüdiger Kapitza
TU Braunschweig

Christof Fetzer
TU Dresden

Peter Pietzuch
Imperial College London

Abstract

Trusted execution support in modern CPUs, as offered by Intel SGX *enclaves*, can protect applications in untrusted environments. While prior work has shown that legacy applications can run in their entirety inside enclaves, this results in a large trusted computing base (TCB). Instead, we explore an approach in which we *partition* an application and use an enclave to protect only security-sensitive data and functions, thus obtaining a smaller TCB.

We describe **Glamdring**, the first source-level partitioning framework that secures applications written in C using Intel SGX. A developer first annotates security-sensitive application data. Glamdring then automatically partitions the application into untrusted and enclave parts: (i) to preserve data confidentiality, Glamdring uses *dataflow analysis* to identify functions that may be exposed to sensitive data; (ii) for data integrity, it uses *backward slicing* to identify functions that may affect sensitive data. Glamdring then places security-sensitive functions inside the enclave, and adds runtime checks and cryptographic operations at the enclave boundary to protect it from attack. Our evaluation of Glamdring with the Memcached store, the LibreSSL library, and the Digital Bitbox bitcoin wallet shows that it achieves small TCB sizes and has acceptable performance overheads.

1 Introduction

Applications are increasingly deployed in potentially untrusted third-party data centres and public cloud environments such as Amazon AWS [3] and Microsoft Azure [4]. This has a major impact on application security [1]: applications must protect sensitive data from attackers with privileged access to the hardware or software, such as system administrators. Applications that rely on cryptographic techniques to protect sensitive data [60, 63, 82] limit the operations that can be carried out; fully homomorphic encryption [32] allows arbitrary operations but adds substantial overhead.

A new direction for securing applications in untrusted environments is to use *trusted execution* mechanisms offered by modern CPUs such as Intel's Software Guard Extensions (SGX) [42]. With Intel SGX, user code and data are protected as part of secure *enclaves*. An enclave is a separate memory region that is encrypted transparently by the hardware and isolated from the rest of the system, including higher-privileged system software.

Haven [6], Graphene [55, 81] and SCONE [2] have demonstrated the feasibility of executing entire applications inside enclaves by adding sufficient system support, such as a library OS or the C standard library, to the enclave. By placing all code inside the enclave, these approaches, however, have a large *trusted computing base* (TCB) that violates the *principle of least privilege* [67]: all enclave code executes at a privilege level that allows it to access sensitive data. An attacker only needs to exploit one vulnerability in the enclave code to circumvent the security guarantees of trusted execution [78]. The number of bugs even in well-engineered code is proportional to the size of the code [54].

To partially mitigate this problem, proposals for securing applications with enclaves [68, 72, 73] introduce additional checks in enclave code to prevent it from compromising the confidentiality or integrity of enclave data. Such approaches, however, restrict the allowed behaviour of enclave code, e.g. prohibiting general enclave code from interacting with memory outside of the enclave [68]. This limits the applicability of trusted execution mechanisms for arbitrary applications.

We want to explore a different design point for securing applications with trusted execution by placing *only* security-sensitive functions and data inside the enclave. We exploit the observation that only a subset of all application code is security-sensitive [11, 71, 74], and ask the question: **"what is the *minimum* functionality of an application that must be placed inside an enclave to protect the confidentiality and integrity of its security-sensitive data?"** Our goal is to develop a principled approach that (i) *partitions* applications into security-

sensitive enclave and security-insensitive non-enclave parts; (ii) gives *guarantees* that the security-sensitive enclave code cannot violate the confidentiality or integrity of sensitive enclave data, even under attack; and (iii) has an acceptable *performance* overhead despite the limitations of current SGX implementations [16].

In our approach, we use *static program analysis* to identify a security-sensitive subset of the application code. Being conservative, it allows us to robustly identify the subset of functions that may be exposed to or modify sensitive data. This analysis is independent of application input, which may be controlled by an attacker, and thus is resilient against attacks on the enclave interface, as long as the assumptions made by the static analysis are enforced at runtime.

We describe **Glamdring**, a new framework for securing C applications using Intel SGX. Glamdring partitions applications at the source code level, minimising the amount of code placed inside an enclave. To partition an application, a developer first annotates input and output variables in the source code that contain sensitive data and whose confidentiality and integrity should be protected. Glamdring then performs the following steps:

(1) Static dataflow analysis. To prevent disclosure of sensitive data, functions that may potentially access sensitive data must be placed inside the enclave. Glamdring performs *static dataflow analysis* [65] to detect all functions that access sensitive data or data derived from it. It tracks the propagation of sensitive data through the application, starting with the annotated inputs.

(2) Static backward slicing. To prevent an attacker from compromising the integrity of sensitive output data, functions that update sensitive data must be placed inside the enclave. Here Glamdring uses *static backward slicing* [84], starting from the set of annotated output variables, to identify functions that can affect the integrity of this data. It creates a backward slice with all source code that the sensitive output variables depend on.

(3) Application partitioning. Glamdring now partitions the application by placing all of the security-sensitive functions identified above inside the enclave. This creates an *enclave boundary interface* that constitutes all parameters passed to enclave functions and accesses to untrusted global variables. Any sensitive data that crosses the enclave interface is transparently encrypted and signed by the enclave code or trusted remote client, respectively. For performance reasons, some security-insensitive functions may be moved inside the enclave.

(4) Source code generation. Finally, Glamdring transforms the application using a source-to-source compiler based on the LLVM/Clang compiler toolchain [14, 49]. It (i) generates appropriate entry/exit points at the enclave boundary with the required cryptographic operations; (ii) ensures that memory allocations for data structures are performed inside or outside of the enclave depending on the nature of the data; and (iii) adds runtime checks at the enclave boundary to ensure that the invariants required for the soundness of the static analysis hold. The output of this phase is an untrusted binary and a trusted shared library that executes inside the enclave.

We evaluate the security and performance properties of Glamdring by applying it to three applications: the Memcached key/value store [24], the LibreSSL library [7], and the Digital Bitbox bitcoin wallet [70]. Our experiments show that Glamdring creates partitioned versions of these applications with TCBs that contain 22%–40% of the lines of code of the applications. Despite their strong security guarantees, the partitioned applications execute with between 0.3×–0.8× of the performance of the original versions.

2 Background

Protecting application data is crucial. Past incidents have shown that data breaches [41] and integrity violations [75] can have a major impact on users [30] and the reputation of application providers [59].

Today applications are deployed frequently in untrusted environments such as public clouds, controlled by third-party providers. In addition to the application being vulnerable, the underlying infrastructure (i.e. the operating system (OS) and hypervisor) may be untrusted by the application owner, and software-based solutions implemented as part of the OS [17,46] or hypervisor [13,20,39] cannot protect application data.

New hardware security features, such as Intel SGX, offer a solution through a *trusted execution* model. It supports memory and execution isolation of application code and data from the rest of the environment, including higher-privileged system software. In this work, we address the problem of how developers can protect only the security-sensitive code and data of an application using trusted execution.

2.1 Threat model

We consider code to be security-sensitive if it accesses sensitive data directly or can impact the confidentiality or integrity of data indirectly. For example, in the Memcached [24] store, assuming that key/value pairs are sensitive, functions that store key/value pairs are security-sensitive, while ones for network handling are not.

The adversary's goal is to either disclose confidential data or damage its integrity. We consider a powerful and active adversary, such as a malicious system administrator, who has control over the hardware and software of the machine executing the application. The adversary may therefore (i) access or modify any data in memory or disk; (ii) view or modify the application code; and (iii) modify the OS or other system software.

We do not consider denial-of-service (DoS) attacks—

an adversary with full control over the machine can decide to not run the application. Such attacks can be detected and potentially mitigated using replication [21]. Similar to other work, we also ignore side-channel attacks that exploit timing effects [83] or page faults [86], but there exist dedicated mitigation strategies [10, 19].

2.2 Trusted execution with Intel SGX

Intel's *Software Guard Extensions* (SGX) [42] allow applications to protect the confidentiality and integrity of code and data, even when an attacker has control over all software (OS, hypervisor and BIOS) and physical access to the machine, including the memory and system bus.

SGX provides applications with a trusted execution mechanism in the form of secure *enclaves*. Enclave code and data reside in a region of protected memory called the *enclave page cache* (EPC). Only application code executing inside the enclave is permitted to access the EPC. Enclave code can access the memory outside the enclave. An on-chip *memory encryption engine* encrypts and decrypts cache lines in the EPC that are written to and fetched from memory. As enclave code is always executed in user mode, any interaction with the OS through system calls, e.g. for network or disk I/O, must execute outside of the enclave.

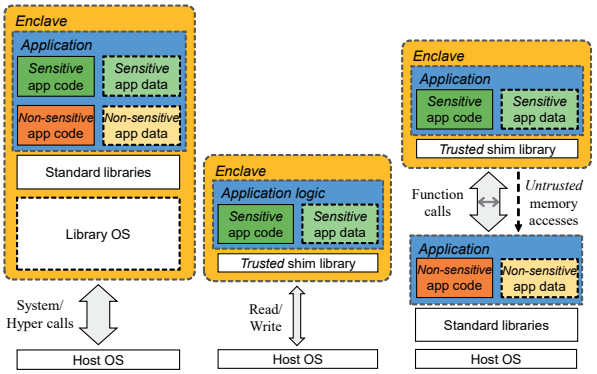
Using Intel's SGX SDK [43], developers can create *enclave libraries* that are loaded into an enclave and executed by a CPU with SGX support. A developer defines the interface between the enclave code and other, untrusted application code: (i) a call into the enclave is referred to as an *enclave entry call* (*ecall*). For each defined *ecall*, the SDK adds instructions to marshal parameters outside, unmarshal the parameters inside the enclave and execute the function; and (ii) *outside calls* (*ocalls*) allow enclave functions to call untrusted functions outside. Added SDK code leaves the enclave, unmarshals the parameters, calls the function, and re-enters the enclave.

Any *ecalls* and *ocalls* introduce a performance overhead because the hardware must perform certain actions to maintain the security guarantees of SGX. Enclave code must also verify the integrity of accessed data, such as parameters of *ecalls*, return values of *ocalls*, and data read from untrusted memory.

2.3 Security with trusted execution

Next we explore the design space for securing application data using trusted execution and discuss the trade-offs with respect to (i) the size of the TCB; (ii) the complexity of the enclave interface; (iii) the development effort; and (iv) the generality of the approach.

With Intel SGX, the TCB consists of the enclave code and the trusted hardware. Following the principle of least privilege [67], only the parts of an application that require access to sensitive data should be executed within an enclave. As studies have shown [54, 69], the number of



(a) Complete enclave interface (b) Predefined enclave interface (c) Application-specific interface

Figure 1: Design alternatives for the use of enclaves

software bugs, and thus potential security vulnerabilities, increases proportionally with the code size. This makes it important to minimise the size of the TCB.

The complexity of the enclave interface, however, impacts the security of enclave code and data. For example, security-sensitive application code inside the enclave must still interact with the untrusted environment to perform I/O. Return values from system calls must be checked to protect against *Iago attacks* [12], in which an attacker compromises the OS kernel to force enclave code to disclose or modify sensitive enclave data. Creating a principled enclave interface makes it easier to reason about the infeasibility of particular attacks.

Important factors that determine the adoption of a given approach for securing applications with secure enclaves are the development effort and whether it is generally applicable to any application. Fig. 1 shows three design alternatives for protecting applications using secure enclaves:

Complete enclave interface. As shown in Fig. 1a, the approach adopted by systems such as Haven [6], SCONE [2] and Graphene [55, 81] provides isolation at a coarse granularity by executing a *complete* application inside an enclave. Haven runs unmodified Windows applications using the Drawbridge library OS [61]; Graphene uses a library OS in the enclave to run Linux applications; and SCONE places a modified version of the standard C library in the enclave for supporting recompiled Linux applications. Both security-sensitive and insensitive application code and data reside within the enclave, increasing the TCB size.

The enclave interface supports a complete set of system/hyper calls, which cannot be handled inside the enclave. The interface is application-independent, but its complexity (in terms of number of distinct calls and their input parameters) depends on the adopted system abstraction. The required system support within the enclave further adds to the TCB size.

While this approach incurs low development effort,

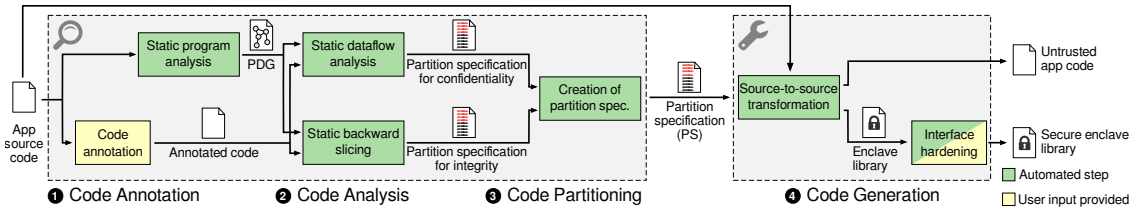


Figure 2: Overview of the Glamdring framework

as it can execute mostly unmodified applications, and is generic across applications, it cannot mask fundamental limitations of SGX when trying to provide a complete enclave interface. For example, SCONE cannot support applications that use the `fork()` system call.

Predefined enclave interface. Fig. 1b shows an approach in which applications must adhere to a *predefined* restricted enclave interface [68, 72, 73]. For example, VC3 [68] protects map/reduce jobs using enclaves and forces map/reduce tasks to interact with the untrusted environment only through a particular interface. The enclave contains a small trusted shim library, resulting in a smaller TCB compared to the previous approach.

This approach results in a minimal enclave interface—VC3's interface consists of only two calls, one to read encrypted key/value pairs and another to write them as the job output. This limited interaction of the enclave with the outside simplifies protection: it is possible to add dynamic checks that enforce security invariants [72], e.g. preventing enclave code from accessing untrusted memory except through the enclave interface.

The security benefits of this approach are offset by its limited applicability. Given the predefined enclave interface, the approach can only be used with applications that interact with the untrusted environment in specific ways, such as map/reduce tasks.

Application-specific enclave interface. We explore another design point. We exploit the fact that, for many applications, only a subset of code handles sensitive data, while other code is not security-sensitive and does not need protection [9, 71, 74]. As shown in Fig. 1c, this makes it possible to *partition* the application to reduce the TCB size, leaving code and data that is not security-sensitive outside the enclave.

Past work has shown that partitioning can be done by hand so that complex applications can exploit enclaves [9, 58]. Instead, we want to explore the hypothesis that it is feasible to use *principled* techniques, such as program analysis, to partition applications for secure enclaves, and provide *security guarantees* about the enclave code and its interface to the untrusted environment.

With this approach, the enclave interface now becomes *application-specific*: a set of *ecalls* and *ocalls* is required between trusted and untrusted application code. In contrast to a complete enclave interface, fewer system calls need *ocalls* because application code that is placed outside the enclave can issue system calls directly.

Since application data now also exists outside the enclave, enclave code must be allowed to access untrusted memory. This means that it is no longer possible to prohibit all memory accesses, as with the predefined enclave interface [72]. Instead, it is important to give security guarantees that, despite the richer application-specific enclave interface, the untrusted environment cannot disclose sensitive enclave data or compromise its integrity.

3 Glamdring Design

We present *Glamdring*, a framework for protecting existing C applications by executing security-sensitive code in an Intel SGX enclave. Glamdring targets the following requirements: it must protect the *confidentiality* of sensitive *input data* and the *integrity* of sensitive *output data (R1)*; apply the principle of least privilege, minimising the code that can access sensitive data *(R2)*; automate changes to the application code *(R3)*; and impose an acceptable performance overhead *(R4)*. To achieve these requirements, Glamdring operates in four phases (see Fig. 2):

(1) Code annotation: Glamdring must know which application data is sensitive (R1). The developer provides information about the *sources* (inputs) and *sinks* (outputs) of security-sensitive data by annotating variables whose values must be protected in terms of confidentiality and integrity (§3.1).

(2) Code analysis: Based on the annotated source code, Glamdring identifies a subset of code that is security-sensitive (R2). It uses automatic static program analysis (R3) to find control and data dependencies on security-sensitive data. Glamdring thus obtains the minimal set of statements that either handle confidential data or affects its integrity (R1/R2) (§3.2).

(3) Code partitioning: Next Glamdring creates a *partition specification* (PS) that defines which parts of the code must be protected by the enclave. The PS enumerates the functions, memory allocations and global variables that are security-sensitive based on the program analysis. This defines the *enclave boundary interface* of the partitioned application, which includes *ecalls*, *ocalls*, and direct accesses to untrusted memory (§3.3).

(4) Code generation: Finally, Glamdring uses a source-to-source compiler that, based on the PS, partitions the code into a secure enclave library and untrusted code. The enclave boundary interface is hardened with runtime

checks that enforce invariants on the program state (§4).

3.1 Code annotation phase

The security-sensitive data that must be protected is application-dependent. To identify it, a developer must therefore annotate the source code. Glamdring relies on the fact that security-sensitive data is protected when it is exchanged between a trusted client and the application. For example, if data is received via the network, Glamdring requires the client to encrypt and sign the data. Both the client and the enclave code use symmetric AES-GCM encryption [22]; the key is established upon enclave creation.

When encrypted security-sensitive data reaches the application through a *source*, such as an I/O channel, or leaves the application through a *sink*, a developer must annotate the corresponding variable using a compiler pragma. The annotation `sensitive-source` identifies a variable at a given source code location where security-sensitive data enters the application; analogously, the annotation `sensitive-sink` indicates a variable at which security-sensitive data leaves the application.

For example, for Memcached we assume that the security-sensitive data is the type of command submitted by the client (`get`/`set`) and its associated key/value data. This data is encrypted and signed by the trusted client when sent to the application. Using Glamdring, Memcached then requires two annotations:

```
1  #pragma glamdring sensitive-source(command)
2  static void process_command(conn *c, char *command) {
3    token_t tokens[MAX_TOK];
4    size_t ntokens;
5    ...
6    ntokens = tokenize_command(command,tokens,MAX_TOK);
7    ...
8    process_update_command(c,tokens,ntokens,comm,false);
9    ...
10 }
11
12 #pragma glamdring sensitive-sink(buf)
13 static int add_iov(conn *c, void *buf, int len) {
14   ...
15   m = &c->msglist[c->msgused - 1];
16   m->msg_iov[m->msg_iovlen].iov_base = (void *)buf;
17   ...
18 }
```

An obvious location for the `sensitive-source` annotation might be the socket `read()` call from which a client request is received. However, this would be unnecessarily conservative because it would denote all network data as security-sensitive (and thus encrypted). Instead, the annotation in line 1 marks the content of the parameter `command`, which holds the request command and data, as security-sensitive. The `sensitive-sink` annotation in line 12 specifies that the output buffer for the client response also contains security-sensitive data.

3.2 Code analysis phase

Next the code analysis phase identifies all *security-sensitive* statements in the program that have dependencies on the set of all annotated statements S_A. This combines (a) for *confidentiality*, the set of all statements that are influenced by the ones in S_A; and (b) for *integrity*, the set of all statements that influence the ones in S_A.

Glamdring uses static program analysis to identify all security-sensitive statements. Static analysis is workload-independent and hence makes conservative decisions about dependencies. To ensure that an attacker cannot violate the invariants that static analysis infers from the untrusted code, Glamdring adds runtime checks during code generation (see §4).

Glamdring's analysis uses a *program dependence graph* (PDG) [23], referred to as P, in which vertices represent statements, and edges are both data and control dependencies between statements. PDGs are effective representations for program slicing [40, 56]. Using P, Glamdring finds the set of all security-sensitive statements as follows:

(1) Static dataflow analysis for confidentiality. Given S_A and P, Glamdring uses graph-reachability to find a subgraph P_c of P that contains all statements with a transitive control/data dependence on statements in S_A (i.e. vertices reachable from statements in S_A via edges in P).

For statements in S_A that are annotated as a `sensitive-sink`, Glamdring encrypts/signs the data before the statement inside the enclave, making it unnecessary to perform dataflow analysis from these statements.

(2) Static backward slicing for integrity. Given S_A and P, Glamdring uses static backward slicing to find a subgraph P_i with all statements in P on which statements in S_A have a control/data dependence (i.e. all vertices from which statements in S_A are reachable via P).

For these statements in S_A that are annotated as `sensitive-source`, Glamdring employs client-side encryption of the data, making it unnecessary to perform backwards slicing from these statements.

Finally, the set of all security-sensitive statements S_s is obtained by combining P_c and P_i.

3.3 Code partitioning phase

Although S_s enumerates security-sensitive statements, Glamdring partitions the application at the granularity of functions rather than statements. This makes the enclave boundary coincide with the application's function interface, easing automatic code generation (§4) and minimising the required code changes (R3).

Glamdring produces a *partition specification* (PS) from S_s with the set of security-sensitive functions, memory allocations and global variables to protect:

(i) *functions*: the PS includes all functions whose definitions contain *at least* one statement in S_s;

(ii) *memory allocations*: the PS must identify allocated memory for security-sensitive data. Statements in S_s with calls to `malloc` (or similar) are enumerated in the

PS, and these allocations are placed inside the enclave;

(iii) *global variables*: the PS lists all global variables accessed in statements in S_s, and these are allocated inside the enclave. Special accessor *ecalls* (with checks) are provided to the untrusted code to access these globals if needed. The PS specifies if the global was part of P_c or P_i or both, which determines what type of access (read, write or none) the outside code has.

Enclave boundary relocation (EBR). Glamdring's code analysis phase produces a lower bound on the code that must be inside the enclave to guarantee security. In practice, however, a partitioning may prove costly in terms of performance if program execution must frequently cross the enclave boundary interface. Glamdring improves performance by moving additional functions into the enclave in order to reduce the number of enclave crossings. Using a representative workload and the output of the `gcov` runtime profiling tool [28], Glamdring assigns a cost to each enclave boundary function according to the number of invocations. Up to a configurable threshold, Glamdring adds functions to the enclave. Adding extra functions to the enclave cannot violate the security guarantees of Glamdring, but it does increase the TCB size.

3.4 Discussion

The security guarantees of Glamdring rely on (a) the soundness of the static analysis; (b) the modeling of external library calls whose source code is unavailable; and (c) the correctness of annotations.

Static analysis. To be tractable, static analysis infers invariants on program state based on the source code. These invariants must also hold at runtime, even when the untrusted code is under control of an attacker. As we describe in §4.2, Glamdring ensures this by adding runtime invariant checks to the enclave boundary.

Static pointer analysis is undecidable for C programs [64] and thus fundamentally imprecise [33, 38]. The existence of false positives, however, does not compromise soundness: the partitioning phase may assign more functions to the enclave than necessary, but never excludes security-sensitive functions from the enclave.

Modelling external library calls. Static analyses must model the behaviour of all invoked functions, including those in external libraries with unavailable definitions. A conservative model makes all output parameters dependent on all input parameters and hence upholds the security guarantees; more precise models can consider actual function behaviour to specify dependencies [5, 36].

Annotations. Most static analysis tools for security rely on developer annotations of sources/sinks of security-sensitive data [35, 76]. While these are application-specific, in many cases they are easy to identify, e.g. when they are well-known library functions for I/O channels.

4 Code Generation and Hardening

The code generation phase produces a source-level partitioning of the application based on the partition specification (PS) (§4.1). In addition, it *hardens* the enclave boundary against malicious input, ensuring that the enclave upholds the confidentiality and integrity guarantees for sensitive data (§4.2). The result is a set of enclave and outside source files, along with an enclave specification, which can be compiled using the Intel SGX SDK.

4.1 Code transformation

The code transformation must (a) handle calls into and out of the enclave; and (b) change the allocation, scope and lifetime of variables and functions in the generated enclave and non-enclave versions of the code.

Glamdring provides a *code generator* that relies on the LLVM/Clang compiler toolchain [14, 49] to rewrite the preprocessed C source code. It uses the Clang libraries to parse source code into an abstract syntax tree (AST), and traverses the AST to analyse and modify the source code. In addition to the enclave and outside source files, it produces an interface specification in the *enclave definition language* (EDL) required by the Intel SDK [43]. The code generation proceeds in three steps:

(i) Moving function definitions into the enclave. For each source file, the code generator creates an enclave and an outside version, which contain a copy of the original preprocessed input file. From the enclave version, it removes all functions not listed in the PS; from the outside version, it removes all listed enclave functions.

(ii) Generating *ecalls* and *ocalls*. Based on the set of enclave functions, the code generator identifies the *ecalls* and *ocalls* that are part of the enclave boundary interface. It traverses the direct call expressions in each function: (a) if the caller is an untrusted function and the callee is an enclave function, the callee is made an *ecall*; (b) if the caller is an enclave function and the callee is an untrusted function, the callee is made an *ocall*.[1]

Adding stubs for encryption/decryption. As mentioned in §3.1, the security-sensitive data received from (and returned to) clients is encrypted (and integrity-protected) using a shared AES-GCM key. The code generator adds code to (a) decrypt security-sensitive data entering the enclave at locations annotated as `sensitive-source`, and (b) encrypt the security-sensitive data leaving the enclave at locations annotated as `sensitive-sink`. The application client must be modified to handle the corresponding encryption/decryption operations.

Handling C library functions. Calls to C library functions are handled separately. A subset is supported by the Intel SDK inside the enclave and is handled in a polymor-

[1] Pointers passed outside the enclave are only deep-copied if data in enclave-allocated memory needs to be declassified—the programmer needs to implement this manually.

phic manner: the enclave and untrusted code call their respective versions.[2] For unsupported library functions, e.g. those making system calls, the code generator creates *ocalls* to the corresponding library function linked to the outside code. These *ocalls* violate the enclave boundary identified through static analysis and hence will be hardened with runtime checks (see §4.2).

Handling function pointers as interface arguments. Function pointer arguments to *ecalls* and *ocalls* are special cases because the target function may not exist at the point of invocation of the function pointer. For example, if an *ecall* passes a function pointer targeting a function on the outside, the program will fail when the enclave attempts to call that function pointer directly. Glamdring employs a static function pinter analysis [89] to identify the possible target functions of function pointer arguments passed to *ecalls* and *ocalls*. The code generator then creates *ecalls* or *ocalls* for the target functions and uses a trampoline to jump to the correct one, as shown in the `jump_to_func` function:

```
/* Initialised to func_A and func_B outside */
int (*addrof_func_A)(int); int (*addrof_func_B)(int);

int jump_to_func(int (*fptr)(int), int x) {
  if (fptr==addrof_func_A) return ocall_func_A(x);
  else if (fptr==addrof_func_B) return ocall_func_B(x);
}

int ecall_enclave_func(int (*fptr)(int),int y) {
  return jump_to_func(fptr, y);
}
```

(iii) Handling memory allocation. The code generator also uses the PS to decide which memory allocations must be placed inside the enclave. For the memory allocations listed in the PS, nothing needs to be done because a `malloc` call inside the enclave allocates memory inside; for other memory allocations, a function must allocate memory outside, and the `malloc` is replaced by an *ocall* to the outside. This arises when placing non-sensitive code into the enclave when (i) partitioning at the function instead of statement level; and (ii) moving functions into the enclave using EBR (see §3.3).

4.2 Code hardening

Next we analyse the attack surface of the enclave boundary interface and describe the protection techniques of the code generation phase against attacks (R1).

Interface attacks. The security of the enclave code depends on the inputs that it receives from the enclave interface. An attacker may manipulate the parameters to *ecalls*, the results of *ocalls*, and accesses to globals.

Secure by construction: The enclave code is, by construction, immune to input manipulation attacks. As long as Glamdring's static analysis is sound, it transitively identifies all code that can affect the confidentiality and integrity of security-sensitive data annotated by the developer, placing it inside the enclave (see §3.2).

However, static analysis infers invariants about the possible values of program variables at different program points, permitting it to prune unfeasible program paths from analysis. The soundness of the static analysis therefore depends on these invariants holding at runtime. Any invariant that relates to untrusted code or data may be compromised by an attacker. The following code snippet gives an example of a debug option that is deactivated in the source code:

```
/* Outside code*/
int dump_flag = 0; // Can be modified by attacker.

/* Enclave code */
int ecall_enclave_func(int dump_flag) {
  char* dump_data = malloc(...);
  if(dump_flag == 1)
    memcpy(dump_data, sensitive_data);
  else
    memcpy(dump_data, declassify(sensitive_data));
  write_to_untrusted(dump_data);
}
```

Static analysis infers that the value of `dump_flag` cannot be `1`, making it impossible to take the branch that does not include the `declassify()` call. Since the value of `dump_flag` does not affect the control flow leading to sensitive data release, Glamdring would allocate it outside the enclave. An attacker could set `dump_flag` to any value at runtime, including `1`, to cause data disclosure.

Runtime invariant checks. To prevent such attacks, Glamdring enforces the invariants assumed by the static analysis at runtime. It does this by extracting invariants from the analysis phase and adding them as runtime checks in the code generation phase. Glamdring applies checks on global variables and parameters passed into and out of *ecalls* and *ocalls*. In the above example, Glamdring adds a check `assert(dump_flag == 0)`.

Checks are also applied to *pointers*. The static analysis infers the subset of `malloc` calls that may allocate memory pointed to by each pointer. Glamdring distinguishes between two cases: (a) the analysis infers that a pointer may only point to *untrusted* memory. A runtime check upholds this and any other invariants on pointer aliasing; or (b) the pointer may point to *enclave* memory. Here, Glamdring's invariant checks prevent pointer-swapping attacks (i.e. a trusted pointer being replaced by another trusted pointer): Glamdring instruments the `malloc` calls inferred for that pointer inside the enclave, storing the addresses and sizes of allocated memory. When a trusted pointer is passed to the enclave via an *ecall*, it is checked to ensure that it points to a memory region allocated by one of the statically inferred `malloc` calls for that pointer. This upholds the results of the static pointer analysis at runtime with enclave checks.

[2] Linked calls to the few stateful C library functions (e.g. `strtok`) typically do not span multiple functions, making it unlikely that such calls get partitioned into different regions.

For checks on *global variables* allocated outside, before each use, Glamdring copies the value inside and applies the check to the local copy.

Enclave call ordering attacks. By construction, Glamdring prevents an attacker from subverting the security guarantees by changing the *order* in which *ecalls* are invoked. The transitivity of static analysis ensures that all functions that have a data/control flow dependence relationship (in either direction) with security-sensitive data are placed inside the enclave. Therefore, any change in the ordering of *ecalls* cannot affect the security guarantees as long as the statically-inferred enclave boundary is enforced. The EBR operation does change this boundary, but only by placing extra functions inside, and therefore cannot violate the security guarantees.

Iago attacks. For applications that use C library functions unavailable in Intel SGX SDK, Glamdring adds *ocalls* (see §4.1). The arguments to such *ocalls* may expose security-sensitive data or their results may cause integrity violations, leading to Iago attacks [12]. For these functions, Glamdring enforces statically inferred invariants on the return values at runtime. Further protection could be done similar to I/O shields in SCONE [2].

Replay attacks. An attacker may tamper with the program state assumed by the enclave by replaying previously issued *ecalls*. Glamdring guarantees the freshness of encrypted sensitive data that is passed to *ecalls*. The client affixes a *freshness* counter to security-sensitive data as part of its encryption (see §3.3). The enclave stores the latest freshness counter for each data item, and validates freshness at *ecalls*. After an enclave restart, the freshness counters must be restored to their latest values [77].

Enclave code vulnerabilities. Enclave code may contain vulnerabilities that can be exploited by an attacker. By reducing the amount of code executed in the enclave, Glamdring makes it more feasible to apply existing techniques to discover and rectify bugs such as buffer-overflows [37,48], data races [45] and memory leaks [47].

5 Evaluation

We evaluate Glamdring by applying it to the Memcached key/value store [24], the LibreSSL library [7] and the Digital Bitbox bitcoin wallet [70]. §5.1 describes the security objectives, the source code annotations and the resulting partitioning and its interface. The TCB (LOC) identified by Glamdring varies between 22% and 40%, and the size of the interface between 41–171 *ecalls* and 51–615 *ocalls* for the three applications. §5.2 presents performance results on SGX hardware: the partitioned applications execute with 0.3×–0.8× of the native performance.

Glamdring implementation. Glamdring uses the Frama-C Aluminium [25] static analysis framework, with the "Impact Analysis" [26] and "Slicing" [27] plug-ins and CodeSurfer 3.0.0 [34]. The Glamdring code generator uses LLVM/Clang 3.9 and has approx. 5,000 LOC.

Memcached [24] is a distributed key/value store. It supports several operations: `set(k,v)`, `get(k)`, `delete(k)`, and `increment/decrement(k,i)`. We apply Glamdring to Memcached 1.4.25 that includes libevent 1.4.14 [62], an asynchronous event library. Memcached has 31,100 LOC and 655 functions.

LibreSSL [7] is a fork of the OpenSSL cryptographic library [18], with the goal to provide a simpler and more secure implementation. We apply Glamdring to LibreSSL 2.4.2 to secure its functionality when serving as a certificate authority (CA). LibreSSL has 176,600 LOC and 5,508 functions, which are divided into three libraries, `libcrypto`, `libssl` and `apps/openssl`. We compile LibreSSL without inline assembly because our static analysis does not support it.

Digital Bitbox [70] is a bitcoin wallet designed for high-security USB microcontrollers. It supports: (i) hierarchical deterministic key generation; (ii) transaction signing; and (iii) encrypted communication. We apply Glamdring to Digital Bitbox 2.0.0 with Secp256k1 1.0.0, a cryptographic library, and Yajl 2.1.0, a JSON library. Digital Bitbox has 23,300 LOC and 873 functions.

5.1 Security evaluation

We evaluate the security of the partitioned application in terms of the *TCB size* and the exposed *enclave interface*.

5.1.1 Memcached

Security objectives. We want to protect the integrity and confidentiality of all key/value pairs in an untrusted Memcached deployment, preventing an attacker from reading or modifying the stored key/value data. For this, we use the source code annotations described in §3.1.

Security-sensitive code. Tab. 1 shows that Glamdring places 40% of LOC, 42% of functions and 68% of global variables of Memcached inside the enclave. EBR moves a single additional function into the enclave, reducing the *ocall* crossings by an order of magnitude for `get` and `set` operations. We conclude that a large portion of the Memcached codebase (without libevent) is security-sensitive, as 87% of its functions and 85% of its global variables are assigned to the enclave.

Partitioned architecture. Glamdring places the following Memcached functionality inside the enclave: (i) binary/ASCII protocol handling functions; (ii) slab and cache memory management functions that manipulate the data structures responsible for the internal storage of key/value pairs; and (iii) the hash functions over key/value pairs. The functionality placed outside includes: (i) thread initialization and registration functions; (ii) libevent functions for socket polling and network I/O; and (iii) signal handlers and string utility functions.

Enclave interface. The enclave interface (see Tab. 1) has 41 *ecalls* and 146 *ocalls*. Out of these, 82 *ocalls* are to C

Application	LOC	Functions	Global variables	Security-sensitive LOC	Security-sensitive functions	Security-sensitive global variables	Ecalls	Ocalls	C lib. ocalls	App. ocalls	Ecall crossings per application request			Ocall crossings per application request		
											`get`	`set`		`get`	`set`	
Memcached	31,100	655	119	12,474 (**40%**)	273 (**42%**)	81 (**68%**)	41	146	82	64	1	1		2	2	
Memcached w/o EBR	31,100	655	119		272 (**42%**)	81 (**68%**)	41	147	82	65	1	1		18	34	
Memcached v1.4.25	13,800	247	84		215 (**87%**)	72 (**85%**)										
libevent v1.4.14	17,300	408	35		57 (**14%**)	9 (**26%**)										
												`sign`			`sign`	
LibreSSL	176,600	5,508	1,034	38,291 (**22%**)	918 (**17%**)	163 (**16%**)	171	613	23	312		6,617			110	
LibreSSL w/o EBR	176,600	5,508	1,034		916 (**17%**)	163 (**16%**)	171	615	23	314		16,545			8,235	
libcrypto v2.4.2	124,800	4,550	833		654 (**14%**)	91 (**11%**)										
libssl v2.4.2	24,300	628	42		83 (**13%**)	7 (**17%**)										
apps v2.4.2	27,500	330	159		179 (**54%**)	65 (**41%**)										
											`seed`	`sign`	`random`	`seed`	`sign`	`random`
Digital Bitbox	23,300	873	105	8,743 (**38%**)	365 (**42%**)	55 (**52%**)	114	51	20	31	23	4	7	4	0	0
Digital Bitbox w/o EBR	23,300	873	105		361 (**42%**)	55 (**52%**)	118	55	20	35	3,252	6,937	672	59	12	11
Digital Bitbox v2.0.0	7,900	382	81		195 (**51%**)	48 (**60%**)										
Secp256k1 v1.0.0	12,900	112	9		52 (**46%**)	1 (**11%**)										
Yajl v2.1.0	2,500	379	15		114 (**30%**)	6 (**40%**)										

Table 1: TCB sizes, enclave interfaces and enclave crossings for Glamdring applications (Application requests are: (i) `get`, `set` for Memcached; (ii) `sign` for LibreSSL; and (iii) `seed`, `sign`, `random` for Digital Bitbox.)

library functions unavailable inside the enclave; 64 *ocalls* are to application functions.

To protect the security-sensitive data between the Memcached client and the enclave interface, Glamdring encrypts the following parameters at the client for each request: (i) the `operation` to perform; (ii) the `key`; and (iii) the `value`. The `keys`, `values` and the request `outcome` are encrypted in the client response.

5.1.2 LibreSSL

Security objectives. Our goal is to protect the confidentiality of the private key of the root certificate of the LibreSSL CA. We annotate the private key as follows:

```
int ca_main(int argc, char** argv) {
  ...
  #pragma glamdring sensitive-source(pkey)
  pkey = load_key(bio, keyfile, keyform, 0, key, "...");
  ...
}
```

Security-sensitive code. Tab. 1 shows that Glamdring places 22% of LOC, 17% of functions and 16% of global variables inside the enclave. EBR moves 2 functions into the enclave, thereby: (i) more than halving the number of *ecall* crossings; and (ii) reducing the number of *ocall* crossings by an order of magnitude for `sign` requests. The majority of functions and global variables assigned to the enclave originate from the libcrypto library, which contains most of the certificate signing logic.

Partitioned architecture. Glamdring places only a subset of LibreSSL into the enclave: (i) the entropy/random number generator; (ii) the RSA and Big Numbers module; and (iii) the X509 module, which stores the certificates. The functionality placed outside includes: (i) the TLS/SSL modules for secure communication; (ii) digest algorithms (MD5, SHA256); and (iii) cryptographic protocols unrelated to certificate signing (DSA, AES)

Enclave interface. LibreSSL exposes 171 *ecalls* and 613 *ocalls* (see Tab. 1). Out of those, only 23 *ocalls* provide access to C library functions; 49% of *ocalls* provide access to global variables; and the remaining 278 *ocalls* are used to execute outside LibreSSL functions.

Glamdring places the private key of the root certificate and any variables that depend on it inside the enclave. The communication between the client requesting a certificate signature and the enclave involves: (i) reading the certificate to be signed; and (ii) outputting the signature. We assume that the root certificate and its private key are given to the enclave during initialisation [44]. Since the signed certificate is not confidential, no explicit declassification is needed before writing it to disk via an *ocall*.

5.1.3 Digital Bitbox

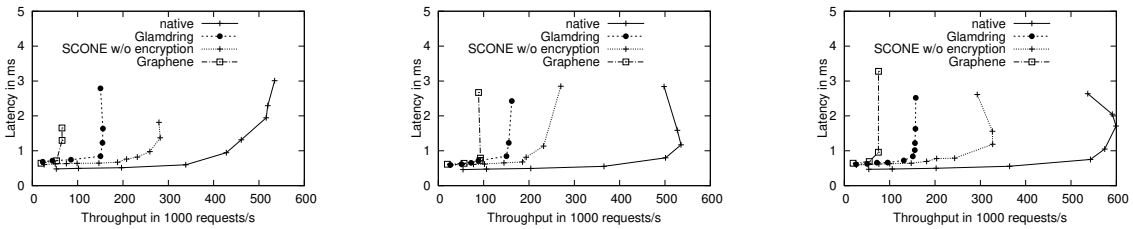
Security objectives. We want to secure Digital Bitbox in a remote deployment, such as an online bitcoin service. An attacker must not (i) read/modify the private keys in the wallet; and (ii) issue commands such as transactions.

We consider three API calls security-sensitive: (i) `seed()` to create a new wallet; (ii) `sign()` to sign a transaction and return the signature; and (iii) `random()` to return a random number. We annotate these API calls with security annotations. The listing below shows the annotation added to protect the transaction signature returned to the user for the `seed()` API call:

```
int wallet_sign(char *message, char *keypath) {
    uint8_t sig[64];
    ...
    ecc_sign_digest(node.private_key, data, sig)
    ...
    #pragma glamdring sensitive-sink(sig)
    return commander_fill_signature_array(sig, pub_key);
}
```

Security-sensitive code. Glamdring places 38% of LOC, 42% of functions and 52% of global variables inside the enclave (see Tab. 1). EBR increases the TCB by 4 functions, reducing the number of *ecall* and *ocall* crossings at runtime by between 1 and 3 orders of magnitude, for the `seed`, `sign` and `random` API calls. Only half of the Digital Bitbox code itself is security-sensitive: 51% of functions and 60% of global variables.

Partitioned architecture. The Digital Bitbox functionality placed inside the enclave includes: (i) command processing functions for specific API calls; (ii) code for generating seeds (using the SGX-provided hardware random



(a) Read-only workload (b) Write-only workload (c) Read/write workload

Figure 3: Throughput versus latency for Memcached native, with SCONE and with Glamdring

Secure application approaches	LOC	Binary Size	Throughput
Memcached with Glamdring	42,800	770 kB	160 kreq/s
Security-sensitive Memcached	12,450		
Glamdring code generation & hardening	5,662		
Intel SGX SDK	24,688		
Memcached with SCONE	149,298	3.3 MB	270–330 kreq/s
Memcached	28,807		
Musl lib. C	105,885		
Stunnel (network encryption)	14,606		
Memcached with Graphene	746,716	4.1 MB	65–95 kreq/s
Memcached	28,807		
Graphene	693,221		
Intel SGX SDK	24,688		

Table 2: TCB sizes and performance for Memcached for Glamdring, SCONE and Graphene

generator); and (iii) elliptic curve operations for transaction signing. The functionality placed outside includes: (i) wallet management functions for retrieving the public key and address formats; (ii) the command interface for handling API calls and constructing responses; and (iii) elliptic curve and JSON parsing utility functions.

Enclave interface. Digital Bitbox exposes 114 *ecalls* and 55 *ocalls* (see Tab. 1). 36% of *ocalls* are to C library functions unavailable inside the enclave; 64% are to application functions outside the enclave.

To protect the security-sensitive data between the client and the application, Glamdring encrypts: (i) the command to execute (`seed()`/`sign()`); (ii) the user-provided entropy for `seed()`; (iii) the transaction data for `sign()`; (iv) the value of `seed()`; (v) the signature of `sign()` returned to the client; and (vi) the generated random number. Performing data protection at this granularity prevents an attacker from issuing commands to Digital Bitbox, and permits Glamdring to move the majority of the JSON parsing functions outside the enclave, as only a subset of the API request/response is security-sensitive.

5.1.4 Discussion

Our security evaluation has led to several insights: First, Glamdring achieves small enclave sizes, protecting security-sensitive functionality for real-world applications. Tab. 2 compares the TCB for Memcached of Glamdring with SCONE [2] and Graphene [55, 81], which place the whole application inside the enclave. As can be seen, Glamdring is one-third the size of SCONE, and one order of magnitude smaller than Graphene in terms of enclave LOC; around 6,000 LOC are added by Glamdring to the TCB through the code generator and enclave interface hardening. In binary sizes, Glamdring is 4× and 5× smaller than SCONE and Graphene, respectively.

Second, EBR is effective at reducing the number of *ecall* and *ocall* crossings at runtime, despite only moving a few additional functions into the enclave. In the case of Digital Bitbox, moving four functions into the enclave reduces the number of enclave boundary crossings by up to three orders of magnitude.

5.2 Performance evaluation

We evaluate the performance of the three partitioned applications in terms of throughput and latency.

Experimental set-up. All experiments are executed on an SGX-supported 4-core Intel Xeon E3-1280 v5 at 3.70 GHz with 64 GB of RAM, running Ubuntu 14.04 LTS with Linux kernel 3.19 and the Intel SGX SDK 1.7. We deactivate hyper-threading and compile the applications using GCC 4.8.4 with `-O2` optimisations.

Application benchmarks. We evaluate Memcached with the YCSB benchmark [15]. Clients run on separate machines connected via a Gigabit network link. We increase the number of clients until the server is saturated. Memcached is initialised with the YCSB default of 1000 keys with 1 KB values. We then vary the percentage of `get` (read) and `set` (write) operations.

For LibreSSL, we measure the throughput and latency when signing certificates using SHA-256 and a 4096-bit RSA key. For Digital Bitbox, we observe the performance for the `seed`, `sign`, and `random` API calls using workloads from the Digital Bitbox test suite: (i) `tests_sign` seeds a wallet and signs 64-byte transactions; (ii) `tests_aes_cbc` seeds a wallet with user-provided entropy, sets passwords and performs encryption/decryption with AES-256; and (iii) `tests_random` returns random numbers.

Results. We measure the throughput and latency for Memcached: (i) partitioned by Glamdring; (ii) executed by SCONE (without network encryption); (iii) by Graphene; and (iv) natively, as the request rate is increased. We consider three workloads: read-only, write-only and 50%/50% read/write.

Fig. 3 shows that all three variants exhibit consistent behaviour across the workloads. Glamdring shows a throughput of 160k requests/s; SCONE (without encryption) achieves between 270k–330k requests/s; Graphene between 65k–95k requests/s; and the native Memcached achieves around 530k–600k requests/sec.

The reason for Glamdring's lower throughput com-

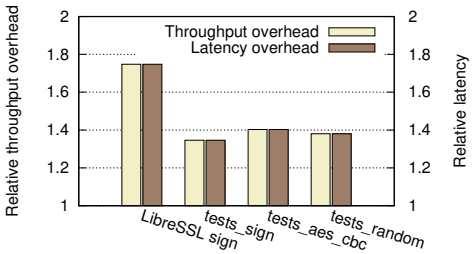


Figure 4: Throughput and latency overhead for Digital Bitbox and LibreSSL

pared to SCONE is that SCONE avoids all enclave transitions; it trades off TCB size for performance (see §2.3) and requires user-level enclave threading to avoid transitions. Enclave transitions dominate the cost of processing a request; by batching multiple `get` requests together using `multi-get`, Glamdring achieves over 210k requests/sec. However, Glamdring has only a third of the TCB of SCONE (see Tab. 2). The overhead of a library OS means that Memcached with Graphene exhibits worse performance than Glamdring.

Fig. 4 shows the performance of LibreSSL and Digital Bitbox with Glamdring compared to their native versions. The throughput of certificate signing in LibreSSL is 0.6× compared to native execution, decreasing from 63 to 36 signatures per second on each CPU core. The performance of LibreSSL is limited by a single *ecall* (`bn_sub_part_words`), which is central to the RSA algorithm and accounts for 95% of all enclave transitions. As for Digital Bitbox, compared to native execution, the relative throughput is between 0.7× and 0.8×; the relative latency is between 1.3× and 1.4×.

Effect of EBR. By comparing the performance of partitioned applications before and after applying the EBR optimisation, we found that the latter increased the throughput by 1.6× to 4.0× for the three applications, at the cost of at most 4 additional functions in the enclave.

6 Related Work

Privilege separation. The attack surface of applications can be reduced in many ways [11, 31, 35, 51, 66, 85]. PrivTrans [11] performs a least-privilege partitioning of an application into a privileged monitor and an unprivileged slave component using static analysis, without considering the integrity of sensitive data. ProgramCutter [85] and Wedge [8] rely on dynamic analysis to partition applications. SeCage [51] combines static and dynamic analysis to partition applications, and the isolation is enforced using CPU virtualisation features. In contrast, Glamdring does not need a trusted OS or hypervisor and respects the constraints of trusted execution.

SOAAP [35] helps developers to reason about the potential compartmentalisation of applications based on source annotations and static analysis. Unlike Glamdring, it does not support automated code partitioning. Rubinov et al. [66] propose a partitioning framework for Android applications. It refactors the source code and adds a set of privileged instructions. However, it only supports type-safe Java applications and requires users to re-implement the security-sensitive functionality in C.

Protecting applications from an untrusted OS. A number of approaches have been proposed to deal with an untrusted OS that spans millions of LOC. NGSCB [57] and Proxos [79] execute both an untrusted and a trusted OS using virtualisation, and security-sensitive applications are managed only by the trusted OS. The TCB, however, still includes a full OS. In more recent work, Overshadow [13], SP^3 [87], InkTag [39] and Virtual Ghost [20] protect application memory from an untrusted OS by extending the *virtual machine monitor* (VMM). Such approaches put trust in the VMM, and cannot protect against attackers with privileged access, such as system administrators.

Trusted hardware. Use of trusted hardware, such as *secure co-processors* [50] and *trusted platform modules* (TPM) [80], can protect against attackers with physical access. A TPM can measure system integrity and provide remote attestation to verify the software stack [29]. Since the TPM measurement will include the OS and any system libraries, the TCB likely comprises millions of LOC.

Flicker [53] reduces the integrity measurement to a TCB of just 250 LOC, but lacks relevant system support and suffers from slow TPM operations. TrustVisor [52] is a special-purpose VMM that uses software-based *µTPMs* for application integrity checking, but it focuses on small pieces of application logic and requires a trusted hypervisor. CloudVisor [88] provides integrity and confidentiality protection for virtual machines using nested virtualisation, but this leads to VM-sized TCBs.

7 Conclusions

We described *Glamdring*, the first partitioning framework that helps developers leverage SGX enclaves for C applications. Glamdring uses static program analysis to decide which subset of the application code to protect, and offers guarantees that the confidentiality and integrity of application data cannot be compromised, even when an attacker has complete control over the machine. Our experimental evaluation demonstrates that Glamdring is sufficiently practical to handle real-world applications.

8 Acknowledgements

This work has received funding from the European Union's Horizon 2020 programme under grant agreements 645011 (SERECA) and 690111 (SecureCloud), and from the UK Engineering and Physical Sciences Research Council (EPSRC) under the CloudSafetyNet project (EP/K008129) and the EPSRC Centre for Doctoral Training in High Performance Embedded and Distributed Systems (HiPEDS) (EP/L016796/1).

References

[1] ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A. D., KATZ, R., KONWINSKI, A., LEE, G., PATTERSON, D., RABKIN, A., STOICA, I., AND ZAHARIA, M. A View of Cloud Computing. *Commun. ACM* (2010).

[2] ARNAUTOV, S., TRACH, B., GREGOR, F., KNAUTH, T., MARTIN, A., PRIEBE, C., LIND, J., MUTHUKUMARAN, D., O'KEEFFE, D., STILLWELL, M. L., ET AL. SCONE: Secure Linux Containers with Intel SGX. In *OSDI* (2016).

[3] Amazon Web Services. https://aws.amazon.com, 2016.

[4] Microsoft Azure. https://azure.microsoft.com, 2016.

[5] BAUDIN, P., FILLIÂTRE, J.-C., MARCHÉ, C., MONATE, B., MOY, Y., AND PREVOSTO, V. ACSL: ANSI C Specification Language, 2008.

[6] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding Applications from an Untrusted Cloud with Haven. In *OSDI* (2014).

[7] BECK, B. LibreSSL–An OpenSSL replacement. The first 30 days, and where we go from here. BSDCAN, 2014.

[8] BITTAU, A., MARCHENKO, P., HANDLEY, M., AND KARP, B. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2008), NSDI'08, USENIX Association, pp. 309–322.

[9] BRENNER, S., WULF, C., LORENZ, M., WEICHBRODT, N., GOLTZSCHE, D., FETZER, C., PIETZUCH, P., AND KAPITZA, R. SecureKeeper: Confidential ZooKeeper using Intel SGX. In *Middleware* (2016).

[10] BRICKELL, E., GRAUNKE, G., NEVE, M., AND SEIFERT, J.-P. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. *IACR Cryptology ePrint Archive* (2006).

[11] BRUMLEY, D., AND SONG, D. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *USENIX Security* (2004).

[12] CHECKOWAY, S., AND SHACHAM, H. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *ASPLOS* (2013).

[13] CHEN, X., GARFINKEL, T., LEWIS, E. C., SUBRAHMANYAM, P., WALDSPURGER, C. A., BONEH, D., DWOSKIN, J., AND PORTS, D. R. Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems. In *ASPLOS* (2008).

[14] clang: a C language family frontend for LLVM. http://clang.llvm.org, 2016.

[15] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking Cloud Serving Systems with YCSB. In *SoCC* (2010).

[16] COSTAN, V., AND DEVADAS, S. Intel SGX Explained. Tech. rep., Cryptology ePrint Archive, 2016.

[17] COWAN, C., BEATTIE, S., KROAH-HARTMAN, G., PU, C., WAGLE, P., AND GLIGOR, V. SubDomain: Parsimonious Server Security. In *LISA* (2000).

[18] COX, M., ENGELSCHALL, R., HENSON, S., LAURIE, B., ET AL. The OpenSSL Project. https://www.openssl.org/, 2002.

[19] CRANE, S., HOMESCU, A., BRUNTHALER, S., LARSEN, P., AND FRANZ, M. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In *NDSS* (2015).

[20] CRISWELL, J., DAUTENHAHN, N., AND ADVE, V. Virtual Ghost: Protecting Applications from Hostile Operating Systems. In *ASPLOS* (2014).

[21] DOULIGERIS, C., AND MITROKOTSA, A. DDoS Attacks and Defense Mechanisms: Classification and State-of-the-art. *Comput. Netw.* (2004).

[22] DWORKIN, M. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. Tech. rep., National Institute of Standards and Technology (NIST), 2007.

[23] FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. The Program Dependence Graph and Its Use in Optimization. *Trans. Program. Lang. Syst.* (1987).

[24] FITZPATRICK, B. Distributed Caching with Memcached. *Linux Journal* (2004).

[25] Frama-C Software Analyzers. http://frama-c.com/what_is.html, 2016.

[26] Frama-C Impact analysis plug-in. http://frama-c.com/impact.html, 2016.

[27] Frama-C Slicing plug-in. http://frama-c.com/slicing.html, 2016.

[28] FREE SOFTWARE FOUNDATION, INC. Gcov - Using the GNU Compiler Collection (GCC). https://gcc.gnu.org/onlinedocs/gcc/Gcov.html#Gcov, 2017.

[29] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., AND BONEH, D. Terra: A Virtual Machine-based Platform for Trusted Computing. In *SOSP* (2003).

[30] GEMALTO NV. No One is Immune to Breaches as 183 Million Accounts Compromised in Q3 2014. https://safenet.gemalto.com/news/2014/q3-data-breaches-compromise-183-million-customer-accounts, 2014.

[31] GENEIATAKIS, D., PORTOKALIDIS, G., KEMERLIS, V. P., AND KEROMYTIS, A. D. Adaptive Defenses for Commodity Software Through Virtual Application Partitioning. In *CCS* (2012).

[32] GENTRY, C. Fully Homomorphic Encryption Using Ideal Lattices. In *STOC* (2009).

[33] GHARAT, P. M., KHEDKER, U. P., AND MYCROFT, A. Flow-and Context-Sensitive Points-To Analysis Using Generalized Points-To Graphs. In *SAS* (2016).

[34] GRAMMATECH, INC. CodeSurfer. https://www.grammatech.com/products/codesurfer, 2016.

[35] GUDKA, K., WATSON, R. N., ANDERSON, J., CHISNALL, D., DAVIS, B., LAURIE, B., MARINOS, I., NEUMANN, P. G., AND RICHARDSON, A. Clean Application Compartmentalization with SOAAP. In *CCS* (2015).

[36] GUTTAG, J. V., AND HORNING, J. J. *Larch: Languages and Tools for Formal Specification*. Springer Science & Business Media, 2012.

[37] HALLER, I., SLOWINSKA, A., NEUGSCHWANDTNER, M., AND BOS, H. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *USENIX Security* (2013).

[38] HARDEKOPF, B., AND LIN, C. Flow-sensitive Pointer Analysis for Millions of Lines of Code. In *CGO* (2011).

[39] HOFMANN, O. S., KIM, S., DUNN, A. M., LEE, M. Z., AND WITCHEL, E. InkTag: Secure Applications on an Untrusted Operating System. In *ASPLOS* (2013).

[40] HORWITZ, S., REPS, T., AND BINKLEY, D. Interprocedural Slicing Using Dependence Graphs. In *PLDI* (1988).

[41] IDENTITY THEFT RESOURCE CENTER. 2016 Breach List. http://www.idtheftcenter.org/images/breach/ITRCBreachReport_2016.pdf, 2016.

[42] INTEL CORP. Software Guard Extensions Programming Reference, Ref. 329298-002US. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, 2014.

[43] INTEL CORP. Intel Software Guard Extensions (Intel SGX) SDK. https://software.intel.com/sgx-sdk, 2016.

[44] JOHNSON, SIMON ET AL. Intel® Software Guard Extensions: EPID Provisioning and Attestation Services. https://software.intel.com/en-us/blogs/2016/03/09/intel-sgx-epid-provisioning-and-attestation-services, 2016.

[45] JULA, H., TRALAMAZZA, D., ZAMFIR, C., AND CANDEA, G. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI* (2008).

[46] KELBERT, F., AND PRETSCHNER, A. A Fully Decentralized Data Usage Control Enforcement Infrastructure. In *ACNS* (2015).

[47] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code-pointer integrity. In *OSDI'14* (2014).

[48] LAROCHELLE, D., AND EVANS, D. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *USENIX Security* (2001).

[49] LATTNER, C., AND ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO* (2004).

[50] LINDEMANN, M., PEREZ, R., SAILER, R., VAN DOORN, L., AND SMITH, S. Building the IBM 4758 Secure Coprocessor. *Computer* (2001).

[51] LIU, Y., ZHOU, T., CHEN, K., CHEN, H., AND XIA, Y. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *CCS* (2015).

[52] MCCUNE, J. M., LI, Y., QU, N., ZHOU, Z., DATTA, A., GLIGOR, V., AND PERRIG, A. TrustVisor: Efficient TCB Reduction and Attestation. In *S&P* (2010).

[53] MCCUNE, J. M., PARNO, B. J., PERRIG, A., REITER, M. K., AND ISOZAKI, H. Flicker: An Execution Infrastructure for TCB Minimization. In *Eurosys* (2008).

[54] MISRA, S. C., AND BHAVSAR, V. C. *Relationships Between Selected Software Measures and Latent Bug-Density: Guidelines for Improving Quality*. Springer Berlin Heidelberg, 2003, p. 724732.

[55] OSCARLAB. Graphene-SGX. https://github.com/oscarlab/graphene, 2017.

[56] OTTENSTEIN, K. J., AND OTTENSTEIN, L. M. The program dependence graph in a software development environment. *SIGPLAN Not. 19*, 5 (Apr. 1984), 177–184.

[57] PEINADO, M., CHEN, Y., ENGLAND, P., AND MANFERDELLI, J. NGSCB: A Trusted Open System. In *S&P* (2004).

[58] PIRES, R., PASIN, M., FELBER, P., AND FETZER, C. Secure Content-Based Routing Using Intel Software Guard Extensions. In *Middleware* (2016), ACM.

[59] PONEMON INSTITUTE. The Aftermath of a Mega Data Breach: Consumer Sentiment, 2014.

[60] POPA, R. A., REDFIELD, C. M. S., ZELDOVICH, N., AND BALAKRISHNAN, H. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *SOSP* (2011).

[61] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. Rethinking the Library OS from the Top Down. In *ASPLOS* (2011).

[62] PROVOS, N., AND MATHEWSON, N. libevent - An event notification library. http://libevent.org/, 2003.

[63] PUTTASWAMY, K. P. N., KRUEGEL, C., AND ZHAO, B. Y. Silverline: Toward Data Confidentiality in Storage-intensive Cloud Applications. In *SOCC* (2011).

[64] RAMALINGAM, G. The Undecidability of Aliasing. *TOPLAS* (1994).

[65] REPS, T., HORWITZ, S., AND SAGIV, M. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL* (1995).

[66] RUBINOV, KONSTANTIN AND ROSCULETE, LUCIA AND MITRA, TULIKA AND ROYCHOUDHURY, ABHIK. Automated Partitioning of Android Applications for Trusted Execution Environments. In *ICSE* (2016).

[67] SALTZER, J. H., AND SCHROEDER, M. D. The protection of information in computer systems. *Proceedings of the IEEE* (1975).

[68] SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., AND RUSSINOVICH, M. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *S&P* (2015).

[69] SHEN, V. Y., YU, T.-J., THEBAUT, S. M., AND PAULSEN, L. R. Identifying Error-Prone softwareAn Empirical Study. *Trans. Softw. Eng.* (1985).

[70] SHIFT DEVICES AG. Digital Bitbox. https://github.com/digitalbitbox/mcu, 2016.

[71] SINGARAVELU, L., PU, C., HÄRTIG, H., AND HELMUTH, C. Reducing TCB Complexity for Security-sensitive Applications: Three Case Studies. In *EuroSys* (2006).

[72] SINHA, R., COSTA, M., LAL, A., LOPES, N. P., RAJAMANI, S., SESHIA, S. A., AND VASWANI, K. A Design and Verification Methodology for Secure Isolated Regions. In *PLDI* (2016).

[73] SINHA, R., RAJAMANI, S., SESHIA, S., AND VASWANI, K. Moat: Verifying Confidentiality of Enclave Programs. In *CCS* (2015).

[74] SMITH, S. F., AND THOBER, M. Refactoring Programs to Secure Information Flows. In *PLAS* (2006).

[75] SOFTPEDIA. Hackers Modify Water Treatment Parameters by Accident. http://news.softpedia.com/news/hackers-modify-water-treatment-parameters-by-accident-502043.shtml, 2016.

[76] STEFAN, D., YANG, E. Z., MARCHENKO, P., RUSSO, A., HERMAN, D., KARP, B., AND MAZIÈRES, D. Protecting Users by Confining JavaScript with COWL. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014), OSDI'14, USENIX Association, pp. 131–146.

[77] STRACKX, R., AND PIESSENS, F. Ariadne: A Minimal Approach to State Continuity. In *USENIX Security* (2016).

[78] SYNOPSYS, INC. Coverity Scan - Open Source Report 2014. http://go.coverity.com/rs/157-LQW-289/images/2014-Coverity-Scan-Report.pdf, 2014.

[79] TA-MIN, R., LITTY, L., AND LIE, D. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *OSDI* (2006).

[80] TRUSTED COMPUTING GROUP. TPM Main Specification v1.2, rev 116 . http://www.trustedcomputinggroup.org/tpm-main-specification/, 2011.

[81] 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.

[82] TU, S., KAASHOEK, M. F., MADDEN, S., AND ZELDOVICH, N. Processing Analytical Queries over Encrypted Data. *Proc. VLDB Endow.* (2013).

[83] WEICHBRODT, N., KURMUS, A., PIETZUCH, P., AND KAPITZA, R. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *ESORICS* (2016).

[84] WEISER, M. Program Slicing. In *ICSE* (1981).

[85] WU, Y., SUN, J., LIU, Y., AND DONG, J. S. Automatically Partition Software into Least Privilege Components Using Dynamic Data Dependency Analysis. In *ASE* (2013).

[86] XU, Y., CUI, W., AND PEINADO, M. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P* (2015).

[87] YANG, J., AND SHIN, K. G. Using Hypervisor to Provide Data Secrecy for User Applications on a Per-page Basis. In *VEE* (2008).

[88] ZHANG, F., CHEN, J., CHEN, H., AND ZANG, B. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *SOSP* (2011).

[89] ZHANG, W., AND ZHANG, Y. *Lightweight Function Pointer Analysis*. Springer International Publishing, 2015, p. 439453.

High-Resolution Side Channels for Untrusted Operating Systems

Marcus Hähnel [*, †]
TU Dresden
mhaehnel@tudos.org

Weidong Cui
Microsoft Research
wdcui@microsoft.com

Marcus Peinado
Microsoft Research
marcuspe@microsoft.com

Abstract

Feature-rich mass-market operating systems have large trusted computing bases (TCBs) and a long history of vulnerabilities. Systems like Overshadow, InkTag or Haven attempt to remove the operating system (OS) from the TCB of applications while retaining its functionality. However, the untrusted OS's control of most physical resources puts it in a much better position to launch side-channel attacks than traditional unprivileged side-channel attackers. Initial attacks focused on the page-fault channel, demonstrating significant information leakage for three *legacy* applications.

We present two new side channels for an untrusted OS which use timer interrupts and cache misses to achieve higher temporal and spatial resolution than the page-fault channel. We leverage the untrusted OS's control over hardware to reduce noise in the side channels to enable successful attacks in just a single run of the target. We demonstrate that our side channels enable attacks against *new* SGX applications such as VC3 that were designed *not* to trust the OS. We also show a new attack against `libjpeg` that extracts images with two orders of magnitude more information than the page-fault channel attack.

1 Introduction

Traditionally, the operating system (OS) protects the integrity and confidentiality of application data and is considered part of the trusted computed base (TCB) of an application. However, decades of experience have shown that it is extremely hard to protect large, feature-rich and widely deployed commodity operating systems. The emergence of cloud hosting services has added the new threat of adversarial cloud operators.

Recently, systems like Overshadow [14], InkTag [27] and Haven [9] were proposed to change the protection paradigm by excluding the OS from the TCB and directly protecting applications. These systems use a trusted hypervisor or hardware to provide applications with memory that is protected from the untrusted OS and with a controlled mechanism for transferring control between applications and the untrusted OS. Haven can also protect applications from Iago attacks [13].

However, Xu *et al.* [44] demonstrate that an adversarial OS can launch deterministic side-channel attacks against protected applications running on SGX. Their attacks can steal documents and outlines of JPEG images from single runs of three legacy applications protected by Haven and InkTag. The attacks are more powerful than traditional side-channel attacks by unprivileged attackers because the untrusted OS retains control of most of the hardware. They exploit the fact that some memory accesses of an application depend on secret data. An adversarial OS observes these accesses by making pages inaccessible in the page table. Each resulting page fault interrupts the application at the moment of the access and reveals the page address to the OS, allowing the adversary to infer secrets that influence the sequence of memory accesses.

Functions within a single code page (e.g., tight loop in *strlen*) and data accesses within a single page (e.g., indexing into small arrays) cannot be observed, as the granularity of the page-fault channel is limited by the 4 KB page size. This imposes a fundamental limit on the temporal and spatial resolution of page-fault based attacks.

In this paper, we present two new side channels that significantly improve the temporal and spatial resolution of attacks launched from an adversarial OS. We demonstrate that new attacks can be launched against applications that are immune to attacks based solely on the page-fault channel. To improve temporal resolution, we use a high-precision timer to approximate *single stepping*. We use a cache side channel to improve the spatial resolution from 4 KB to 64 byte cache lines. Our cache side channel has much higher accuracy than the traditional cache side channel controlled by an unprivileged attacker because the OS controls the hardware. The OS can break into an application right before and after the memory access of interest and reduce cache pollution through its control over scheduling.

To demonstrate the power of our new side channels, we build new attacks against VC3 [39] and `libjpeg` [1]. VC3 is a secure MapReduce framework that protects the confidentiality of distributed MapReduce computations by running them inside SGX enclaves [29]. The unmodified legacy applications attacked in [44] had been written under the assumption that the OS was trusted. However,

[*] Work done in part during an internship at Microsoft Research.

[†] Supported in part by the German Research Foundation (DFG) within the CRC 912 - HAEC.

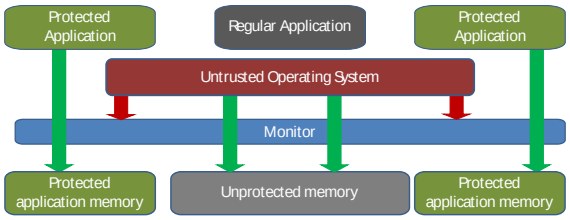


Figure 1: System model: A monitor constrains the OS and prevents it from accessing the protected applications' memory.

VC3 is new code designed to have a small trusted computing base and to run in an SGX enclave on an adversarial OS. Our attack on VC3 is a sequence of attacks that extract various pieces of information from different parts of the VC3 code. None of the individual attacks could have been performed using only the page-fault channel. Our attack on VC3 recovers almost two thirds of the input documents of the WordCount application. Compared to the earlier attack against `libjpeg` [44], our new attack can extract two orders of magnitude more information and recover images with richer detail.

This paper makes the following contributions: We demonstrate

- two new high-resolution side channels for an untrusted OS to attack protected applications;
- a significantly improved attack against `libjpeg` and a new attack against VC3;
- the increased significance of side-channel attacks for untrusted operating systems.

2 System Model

As outlined in Figure 1, the system consists of one or more applications portected by a monitor that constrains the untrusted OS.

The Monitor can be a hypervisor or a part of the CPU. It imposes the following constraints on the OS:

1. It guarantees safe memory to a trusted application. The OS cannot read or write such memory.
2. It controls transitions between an application and the OS, ensuring that the OS can invoke protected code only at well defined entry points. This also allows the monitor to protect the application from leaking CPU state during the transition. On unexpected transitions (traps, interrupts), the monitor saves and scrubs CPU registers.
3. The monitor takes anti side-channel provisions.

We assume the following list of anti side-channel measures. It is based on SGX [28], which, at this time, is the only technology that takes serious measures against side channels.

- The bottom twelve bits of page fault addresses (which specify the offset within the page) are hidden from the OS.
- Hardware debugging facilities (e.g., debug registers, single stepping) are disabled.
- Hardware performance counters are disabled.

The Untrusted OS is considered adversarial—either as a result of malware or because it is under the control of an adversarial administrator. Within the constraints imposed by the monitor, it can take any action to corrupt the application or extract information from it. It has access to all hardware on the system, except as prohibited by the monitor. This includes all unprotected memory, page tables and system devices such as timers.

Applications are protected by the monitor. We target native applications that process secret information and display memory access patterns that depend on the secret information. We assume that at least some parts of the application binaries are public, either because they belong to unmodified legacy applications [9,14,27,44] or because they are a part of a public application platform [39].

3 Background

We describe SGX [29], the page-fault channel [44], and prime-and-probe cache side-channel attacks [36].

3.1 Intel SGX

Intel SGX [6, 26, 29, 34] is a CPU technology recently introduced by Intel. SGX allows the construction of isolated memory regions for applications (*enclaves*) that are protected by the CPU from all other software running on the system, including the OS. In terms of our system model, SGX constitutes a monitor. Enclaves are restricted to running in user mode, but are constructed by the untrusted OS using new privileged instructions. Remote attestation ensures that the owner of the application can detect tampering by the OS before revealing any secrets.

SGX leaves the hardware interface for the OS largely unchanged, enabling compatibility with legacy operating systems, but also providing the OS with a large tool chest for building side channels, such as the page-fault channel [44].

However, SGX includes a number of anti side-channel measures. SGX makes it impossible to use the following CPU features against enclaves: Hardware breakpoint registers (DR0-DR3), single stepping (RFLAGS.TF), Last Branch Record (LBR), Precise Event-Based Sampling (PEBS), and hardware performance counters. Upon an exception or fault inside an enclave, SGX masks the twelve least-significant bits of the faulting address. Thus, the untrusted OS only receives page-granular fault information.

3.2 The Page-Fault Channel

Xu *et al.* describe side-channel attacks based on page faults [44].The attacks take advantage of *input-dependent*

```
size_t strlen(char* str) {
  size_t len = 0;
  while (*str++ != '\0') len++;
  return len;
}
```

(a) `strlen`

```
char* agentNames[] = {"James R. Clapper", "John Doe",
                      /* 510 more entries */  };
char* getAgentName(int agentCode) {
  return agentNames[agentCode];
}
```

(b) Array Access

Figure 2: The secret-dependent memory access does not leak information if the attacker can only observe at page granularity.

memory accesses to infer an application's secret input. For instance, if a global variable is incremented for each user login, an adversary can count the number of accesses to this variable to infer the number of logins. To detect that the global variable is accessed, the adversary can make the page where it resides inaccessible. Any access to the page will trigger a page fault.

In the simplest case, the adversary can use the fault address to decide if the application tried to access the global variable of interest. However, since the offset in the fault address is hidden, the adversary has to infer it. Xu *et al.* propose using page fault *sequences* that can uniquely identify a data access or control transfer.

Attacks based on page-fault sequences still face a fundamental limit: *Page faults only work at page granularity*. The following two examples are immune to attacks via the page-fault channel.

Temporal Limit: Figure 2a shows a simple `strlen` implementation. Assume the binary code is on a single code page and the string `str` is on a single data page. The `len` variable is usually in a register for optimization and thus not usable by an adversary. One instruction in `strlen` reads a character of `str` from the data page. To execute it, the attacker must make both code and data page accessible. But after this, `strlen` execution will not cause further page faults, preventing the adversary from counting the number of iterations and inferring the length of the string.

Spatial Limit: Assume table `agentNames` in Figure 2b is on a single data page. Since the offset of the page-fault address is hidden, an adversary cannot use an access to the table to infer the value of the variable `agentCode`.

3.3 Prime-and-Probe Cache Side-Channel Attacks

Small, fast caches in CPUs are used to mitigate the performance impact of memory accesses. Upon a memory access, the CPU copies the memory contents into the cache. Subsequent accesses to the same address can be serviced from the cache at a much lower cost. This copying takes place at the granularity of 64 byte *cache lines*.

A memory location is mapped to a small group of cache lines based on some of its address bits (bits 6-11 for the Intel L1 data cache used in this paper). The Intel L1 cache uses groups of 8 cache lines (*8-way set associative*). Upon a memory access not currently in the cache, one line in the group must be *evicted* to make space for the new contents.

Caches are typically shared by all code running on a core or even the entire CPU package. An attacker may make a sequence of memory accesses loading the contents into the cache and filling it completely. A subsequent memory access by another program will also cause data to be loaded into the cache, evicting one of the attacker's cache lines.

The attacker can measure the time it takes to access the memory locations he previously loaded into the cache to detect such evictions. Increased access times indicate an eviction, telling the attacker not only that an access took place, but also revealing bits 6-11 of the address of the access.

This procedure is known as *prime and probe* [36]. Filling the cache with the attacker's content is called the *prime* step. Measuring the memory access times is called the *probe* step.

4 Design

This section describes our techniques for performing synchronous, high-resolution side-channel attacks. Abstractly, the attacker has two capabilities: *break*, to set conditional breakpoints on the application and *observe*, to inspect artifacts of its execution (e.g., memory accesses). A technique may also provide both capabilities. Making a page inaccessible [44] allows breaking (page fault) and observing (page fault address).

We introduce two new techniques to overcome the limitations of the page-fault channel (Section 3.2) and significantly broaden the class of application code subject to side-channel attacks:

- A technique to single-step protected applications.
- A cache side channel to observe memory accesses at cache-line granularity.

Both techniques work even if the attacker can observe only a single application run. Thus, any protected application is a potential target for the attack.

We use page-fault sequences to infer memory accesses when possible, allowing us to deploy high-overhead attacks only during the short times they are required.

4.1 Noise reduction

The techniques require a very low level of system noise. We use the OS's control over hardware to reduce noise by disabling interference sources (turbo-boost, prefetching, power management) and preventing preemption of the

victim application. These adjustments can also be made by a compromised OS or a malicious admin.

Like page-fault based attacks [44], our attacks rely on exceptions and interrupts to provide the *break* mechanism. Handlers run on a victims's core to gain access to the private resources (e.g., caches) of the core for *observation*.

4.2 Single Stepping

In our system model (and under SGX), the monitor prevents the OS from using the single stepping features of the hardware. However, the OS can approximate such functionality using hardware timers.

For this work, we use the x86 local APIC timer, due to its high resolution and easy programmability. We use the timer in single-shot mode at the highest available frequency (divider=1). In this mode, the OS writes a target value x into a register. The timer triggers an interrupt x timer ticks after the register write. The timer tick frequency is significantly lower than the TSC frequency. On the Skylake system used in the evaluation, the former is 24 MHz, while the latter is 167 times higher. While this means we can only trigger an interrupt every 167 CPU cycles, we know at which TSC value the interrupt will arrive, allowing us to wait an appropriate time should the desired TSC value be too soon.

Assume we have interrupted the application and want to single-step forward to the next instruction. This may not be easy for all instructions. Here, we focus on instructions with memory operands, which is sufficient for our attacks. The goal is to have the next interrupt arrive during the execution of the next such instruction. If the interrupt arrives during the right time window, the processor will delay the interrupt until execution of the instruction is complete. During regular execution, this time window is extremely short. However, the attacker has an array of tools that can make the instruction very slow and, thus, expand the time window to hundreds of cycles. We only rely on a TLB flush, which causes instructions with memory operands to incur a page-table walk. Additional options include flushing the cache to force page-table walks to incur the full memory latency, manipulating the memory clock to increase memory latency or disabling the cache completely to make all subsequent instructions slow.

The time between starting the timer (writing to the timer register) and executing the victim's next instruction includes the interrupt return path (privilege level change) and the re-entry path of the protected application (crossing the trust boundary, restoring registers). While CPU specific, this time can be measured using a protected application on the attacker-controlled target system. Based on this measurement, we determine the number of timer ticks x. This estimate can only be sufficiently precise if jitter is low, making the noise reduction techniques described above vital.

This mechanism can be used to implement any conditional breakpoint that depends only on attacker-observable information. Single stepping and observing the system after each step allow the attacker to perform a detailed analysis of the system's behavior. While possible for the whole application, it is also quite slow, as each instruction causes at least one interrupt. The overhead can be reduced by using a cheaper breakpoint to narrow down the region of interest (page fault, coarse-grained timer interrupts) and switch to single-stepping there.

We synchronize our single steps with observations about the memory accesses made by the application. For example, the `strlen` code in Figure 2a accesses the string exactly once per iteration. Observing an access to `str` after a single step informs us that the application must be at (close to) the instruction following the access. We obtain the number of loop iterations by counting the number of `str` accesses. Memory accesses are observable by reading the dirty and accessed bits in the page table entries.

4.3 Cache Side-Channel Attack

A cache side-channel allows observations at the granularity of 64 byte cache lines. It has the potential of revealing information about accesses to small (sub-page sized) arrays for which the page-fault channel is ineffective. We use a prime-and-probe attack against the core-local, 32 KB large, 8-way set associative L1 cache.

The key challenge for cache side-channel attacks is *cache pollution* caused by other accesses than the ones of interest during prime and probe. An unprivileged attacker has little control over when his code will execute. Probing may observe the results of both the memory accesses of interest and potentially many unrelated accesses, a problem exacerbated by the small size of the L1 cache.

Traditional cache side-channel attacks mitigate the problem by averaging over many prime-and-probe observations. This technique does not apply to applications that usually execute over each unique input once. Instead, we use our control over hardware to tackle the problem.

Our attack proceeds as follows. We break the application shortly before the memory access we wish to observe, prime the L1 cache, and resume the application. We break again shortly after the access of interest, probe the L1 cache, log the result and resume the application.

Thanks to the OS's control over hardware, we can use page faults or single-stepping to break right before and after the memory access of interest, thus avoiding unrelated memory accesses during prime and probe. To further reduce cache pollution, we prevent other applications from running on the victim's physical core.

Even after applying these mitigation techniques, we still cannot eliminate cache pollution completely for two reasons. First, some code is executed between priming and the resumption of the application and between the interruption of the application and probing. Any memory accesses by such code may result in spurious cache evictions. Second, TLB flushes happen when transitioning into and out of protected applications to protect application memory. This leads to page-table walks when executing subsequent instructions. This in turn causes one memory access per page-table level.

However, we can predict and to some degree even control which cache addresses are being polluted and adjust our evaluation accordingly. Given the 8-way set associativity of the L1 cache, a polluted cache address will also not lose all information about the access we are trying to observe. If the access of interest falls into a cache address polluted by one extraneous access, we should observe L1 misses for two of the eight ways for that address. This allows us to observe the access of interest even in the presence of deterministic cache pollution.

We determine the set of cache lines that will be polluted deterministically on every prime-probe observation. This includes the memory accesses made by the page-table walk for the (known) target page of the access of interest and the (known) page of the page-fault handler, as well as some known accesses made by the handler itself. In our analysis, we subtract these L1 misses from our observations. We call this step *deterministic filtering*.

When probing, we must measure access times very precisely, as L2 hits take only a few cycles longer than L1 hits. We disable interrupts to gain exclusive use of the core for our measurement code.

5 Attacks

We now describe our attacks against VC3 and `libjpeg`.

5.1 VC3

VC3 [39] allows users to run MapReduce jobs [18] in the cloud without exposing their code or data to the provider. VC3 shields computations from the provider using SGX. The provider and all his software (OS, hypervisor) are assumed to be adversarial. Just as in regular MapReduce, the user writes a *map* and a *reduce* function. The functions are encrypted, packaged together with the VC3 framework and sent to the cloud to be run in SGX enclaves. VC3 is designed to plug into an existing untrusted MapReduce framework such as Hadoop [7]. A VC3 job will begin with encrypted input splits that the untrusted MapReduce framework feeds into *mapper* enclaves. Inside a mapper enclave, the VC3 framework decrypts the input and the user's map function, and invokes it. The

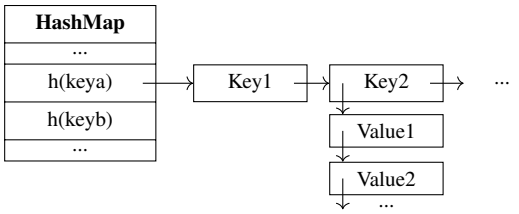


Figure 3: Layout of the hash map used in the VC3 reducers

VC3 framework encrypts intermediate key-value pairs produced by the map function and feeds them into the untrusted MapReduce framework, which distributes them to *reducer* enclaves.

VC3 provides tamper detection for the untrusted communication channel (e.g., removal or duplication of intermediate key-value pairs). Randomized encyption prevents frequency analysis on intermediate key-value pairs. But it also stops the untrusted MapReduce framework from grouping them. Since grouping is required by the *intermediate key-value rule* [18, Sec. 2] VC3 has to implement it inside the reducer enclaves.

The grouping implementation is hash-map based (Figure 3). Each hash key is mapped to an index in an array of 8-byte pointers. Hash collisions are resolved through a linked list of colliding keys for each array index. As the array is 64-byte aligned, eight consecutive 8-byte pointers fill up a cache line. Conversely, observation of a cache line access identifies the corresponding eight array slots. This grouping operation is the only part of VC3 whose memory accesses depend on user data, because most mappers use user data to compute intermediate keys.

5.1.1 Attack Overview

Our attack targets MapReduce applications that have one or more English documents as the input and words in the documents as the intermediate keys. WordCount and Inverted Index are such applications [18, Section 2]. Our goal is to recover as much of the document as possible.

Following the VC3 model, we consider the user's map and reduce functions to be secret, but the VC3 framework to be public, and thus only attack the latter. It would negate the value proposition of VC3 if users were forced to admit code into their TCB that they cannot inspect.

The high-level idea of our attack works as follows. We use our single stepping and cache side channel techniques to infer the length and the (approximate) hash array index for each input word (i.e., intermediate key). We also track words through the various stages of processing in MapReduce to remember their positions in the input document. Finally, we use an English language model that contains a dictionary of words (unigrams) and word pairs (bigrams) as well as their weights (measured by their popularity)

to recover the input document based on the length, hash array index and position of each word.

5.1.2 Word Length

VC3 hashes all intermediate keys (i.e., input words) to insert the key-value pairs into the hash map. The hash function loops over the characters of the key until it finds a null character. The length of the input word is obtained by breaking on the hash function and observing the number of loop iterations it performs using the technique of Section 4.2.

5.1.3 Cache Line Address

We use the cache side channel of Section 4.3 to observe for each word the cache line of the hash array slot into which it is inserted. We break right before the array lookup by making the array pages inaccessible. Upon the page fault, we log the page number, make the page accessible, make the reducer's stack inaccessible and prime the cache. After resuming execution, VC3 accesses the array and, immediately after that, page faults when trying to access its stack. We probe and log the cache state, resolve the page fault, make the array page inaccessible again and resume execution.

5.1.4 Word Position

As input words move through the mappers and reducers, we must keep track of their positions in the input document. Our English language model relies on word order and, more generally, the output of the attack appears much more useful if it presents the words in the correct order.

If there is only a single mapper and a single reducer, the problem is trivial because the words arrive at the reducer (where we observe word lengths and hash slots) in input order. However, multiple mappers and reducers will be sending and receiving intermediate key-value pairs concurrently. Furthermore, VC3 mappers internally determine the reducer for each word, buffer the word and only send buffers containing many words to each of the reducers.

The first problem (concurrency) is easily solved by observing that the attacker controls the communication channel between all mapper and reducer enclaves. In particular, the attacker can observe the order of all messages sent from mappers to reducers. The second problem is more complex. Using the page-fault channel to monitor the mappers, we track for each input word the buffer into which the mapper inserts it and its position in the buffer (details omitted due to space constraints). This information allows us to recover the original word positions when a reducer finally processes the words from the buffer; i.e., when we extract the length and hash slot for each word from the buffer.

5.1.5 Word Recovery

We can represent the information recovered so far as a version of the input document in which each word has been replaced by its length and its hash slot (covering 8 array indices). It remains to map this information back to the original words.

As a first step, we group the words in our language model by length and hash slot. The result is a candidate list for each length, hash slot combination and, thus, for each position in our input document. Next, we refine the candidate list for each input word with the help of context: the bigrams from our language model. For each candidate word at a position, if there is no candidate word in the subsequent position to construct a word pair (bigram) that is contained in our language model, we eliminate the candidate word. We repeat this pruning step on all candidate words iteratively until no candidate words can be removed. Finally, we sort all remaining candidate words for each position based on their weights.

5.2 JPEG

JPEG is one of the most widely used image compression standards. JPEG compression cuts the image into blocks of 8 by 8 pixels and performs a discrete cosine transform (DCT) on each block, followed by other compression steps. JPEG decompression reverses these steps, performing an inverse DCT as one of the last steps. In our attack, we target the `libjpeg` library [1], the most widely used JPEG implementation. Specifically, we exploit the last stage of the inverse DCT function which computes the final values of the 8 by 8 output matrix by means of array lookups. The array has 1024 single byte entries and lies typically on a single page, which makes page-granular observation useless. The final output values are the values read from the array.

Our attack strategy is to observe the cache line accessed in each of the array lookups. However, even at cache-line granularity, we are unable to distinguish between the 64 adjacent array indices that fall into each cache line. For example, if the array is 64 byte aligned, array slots 0 to 63 fall into the same cache line, and our observations do not let us distinguish among them. If the array was filled with random numbers, observing cache lines would be unlikely to reveal useful information.

Fortunately, the array values are either constant or linearly increasing over large ranges. Thus, the average over all array values that lie in the same cache line contains useful information. For cache lines that cover array regions where the values increase linearly, the avarage contains the two most-significant bits of the 8-bit array

values, as we are losing the 6 ($= log_2(64)$) least significant bits due to 64-bit cache-line resolution. For cache lines that cover constant regions, the average contains the same information as the individual array values.

Upon observing an array access at a particular cache line, we use the average over all array values that are covered by that cache line as our inferred output value. We feed these recovered values directly into the last phases of JPEG decompression to obtain the final image. We recover the image dimensions and the color space using the method described in [44].

6 Implementation

This section describes how we implemented the attacks and the target applications. Our prototype was designed for x86-64 PCs running Windows and using Intel SGX as the monitor. The choice of Windows was not essential. A dedicated attacker might choose to write a special attack OS or even build special hardware that gives him easy access to the required functionality. We used the shortcut of adding attack functionality to an existing OS by means of a kernel driver. We chose SGX as the monitor for three reasons. (1) VC3 only runs on SGX. (2) SGX has a detailed, public specification. (3) SGX includes defenses against side-channels, making it a more interesting target.

6.1 Implementation on Windows

We used a Windows driver to implement the techniques from Section 4 and the page fault channel in approximately 1,200 lines of C++ code and 250 lines of assembly code. All binaries were compiled with the Microsoft C/C++ compiler version 18.00.21005.1 with full optimization (`/Ox`) and inlining (`/Ob2`).

The driver hooks the timer handler and the page fault handler in the interrupt descriptor table (IDT). This causes the processor to invoke our handlers, rather than the OS's. Our driver processes all events intended for it and forwards all others to the Windows handlers.

We pinned the target application to a single core and set its scheduling priority to `REALTIME` in order to minimize interference from other OS activity. The core still receives interrupts and other system events which are a source of residual noise. For ease of implementation, we disabled hyper-threading, turbo-boost, pre-fetching and power management in the BIOS. The same can be done in code by the OS.

6.2 SGX

We used the SGX simulator that was used in the evaluation of VC3 [39]. The simulator tries to faithfully implement the essential functionality of SGX in software. For the purposes of our attacks, only the SGX behavior on transitions into and out of enclaves is relevant. In particular, the simulator adds TLB flushes and delay cycles on all transitions. We simulated the saving of the register state by SGX by saving all the registers to a memory page and putting them in a defined state. An equivalent alternative would have been to use the OpenSGX simulator [31].[1]

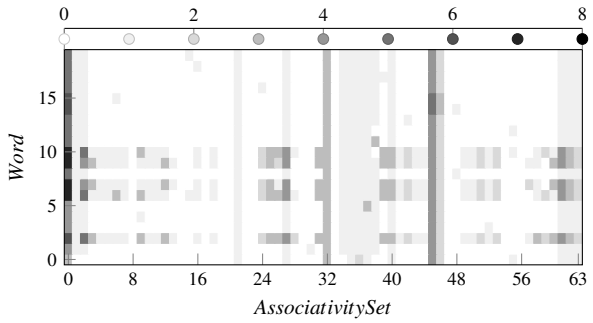


Figure 4: Results of prime-probe observations for 20 distinct words (rows). Darker fields indicate more evicted ways within an 8-way associativity set. Vertical lines identify cache addresses evicted in every observation.

6.3 Single Stepping

As Windows uses the local APIC timer as its system timer we had to share it. We programed the timer, such that it continues to trigger the relatively low-frequency periodic interrupts Windows expects. When single stepping, we set up the timer for single-shot mode with a divider of one.

6.4 Cache Side-Channel Attacks

At initialization, we allocate a page-aligned 32 KB buffer consisting of eight 4KB pages, which correspond to the eight ways of the L1 data cache. Our *prime* procedure loops over the buffer and performs one write operation for each cache line, thus filling the entire L1 data cache.

Our *probe* procedure also iterates over the cache lines covered by our buffer. For each cache line, it times the corresponding memory read using `rdtsc` using serializing and fence instructions to prevent reordering. Accesses that take at most 10 cycles are considered L1 hits.

Our IDT hooks let us control all code executed when transitioning into and out of the victim application. The separation of the L1 cache into an instruction cache and a data cache ensures that code execution by itself does not pollute the L1 data cache. We carefully chose all assembly instructions in our fault handler to control its data memory accesses.

Figure 4 shows the results of 20 prime-probe observations. Each row shows a cache fingerprint for a different

[1] Intel has released an SGX SDK, which only allows enclaves to run in debug mode. This mode disables most protections, thus providing limited additional value over the existing simulator.

word. Thus, we would expect each row to only differ in one gray dot for the accessed cache line. This does not explain all observed differences between the rows. The vertical lines (e.g., at index 32) stem from deterministic cache pollution since they are evicted in every prime-probe observation. We are uncertain as to the exact cause of the noise patterns between rows 5 and 10.

While the pollution is not constant across observations, there appears to be a relatively small number of pollution patterns. For example, lines 6, 7, 9 and 10 have roughly the same pollution pattern. Our strategy was to group our observations by their pollution patterns and to remove cache misses that are common within each group from each of the observations in the group. For this, we computed for each group the average number of observed misses for each cache line and subtracted these averages from our observations, setting negative results to zero. This *noise filtering* removed most of the pollution, significantly reducing the number of candidate cache lines.

6.5 VC3

We used the original VC3 code [39], implemented WordCount [18, Sec. 2.1] in 100 lines of C, and used the VC3 packing tool to generate the binary that is loaded into the enclaves. This tool encrypts the binary containing the mapper and reducer functions and embeds it into a second (plaintext) binary containing the trusted VC3 framework which is later loaded into the enclaves. We used the VC3 preprocessing tool to convert the input document into encrypted input splits for the mappers.

Rather than using Hadoop, we wrote a small program to send inputs into the mappers and reducers and to receive their outputs. This implementation shortcut is valid, since Hadoop is considered untrusted and under attacker control in the VC3 security model. For simplicity, our program runs the mappers and reducers sequentially, storing the intermediate key-value pairs of each mapper on disk.

6.6 JPEG

We built a self-contained application around `libjpeg` that can be run inside an SGX enclave and that decodes an image. We had to provide a small runtime library to satisfy the external dependencies of `libjpeg` (memory allocation, file I/O). We reused the memory allocator from the VC3 runtime and provided just enough file I/O to read the input JPEG file from enclave RAM.

We compiled the unmodified source code of `libjpeg` version 9a, the runtime and the `main` function into a single Windows PE binary without external dependencies. We constructed an enclave consisting of this binary, an encrypted JPEG file (loaded into enclave memory) and heap memory. Upon invocation of the enclave, the `main` function decrypts the file in enclave memory, initializes the runtime and calls `libjpeg` to decompress the image.

cntr. increase	0	1	> 1
mean	64,393,418	204,040,806	1,231
CV	0.1	0.03	0.73

Figure 5: Single stepping experiment: interrupt count by observed counter increase. The mean is taken over 20 repetitions. CV is the coefficient of variation (mean divided by standard deviation).

7 Evaluation

We ran the experiments on a Windows 10 system based on an MSI Z170A motherboard with a 4.0 GHz quadcore Intel i7-6700K Skylake CPU, 8 GB of RAM and a 128 GB SanDisk X300 SSD. We disabled several unnecessary devices (DVD drive, audio, dedicated graphics card).

7.1 Single Stepping

We ran the following microbenchmark to evaluate the effectiveness of single stepping. We set up a victim application that increments an in-memory counter in a tight loop. The compiled loop code consists of three instructions: `add [rdi],1` (increment the counter in memory), `dec rax` (decrement the loop variable) and `jne -9` (conditional jump to the first instruction).

We shared the counter variable with the interrupt handler in our driver and made it record its value at each interrupt. This allowed us to observe the number of loop iterations that the victim had executed between consecutive interrupts. The driver also recorded the address of the interrupted instruction. Before returning, the interrupt handler restarted the timer and flushed the TLB. This experiment requires careful tuning. Space limitations force us to omit many details.

We ran the experiment until $2^{28} \approx 268$ million interrupts had occurred. We repeated the experiment 20 times for a total of more than 5 billion observations. The results are displayed in Fig. 5. About 24% of the interrupts occurred before the next counter increment. They waste cycles, but do not affect accuracy, as we can detect this case in real attacks. More than 99.9993% of the remaining interrupts break into the loop at consecutive iterations. This level of accuracy is more than sufficient for our attacks.

In most cases, the interrupt occurred directly after the `add [rdi],1` instruction. The delay due to the page-table walk for the memory operand (caused by the TLB flush) appears sufficient to absorb most of the timer jitter. We repeated the experiment, but invalidated only the application's TLB mapping to the counter instead of the entire TLB. This change did not have a significant im-

Figure 6: Accuracy of the cache observations of random array accesses with and without deterministic filtering.

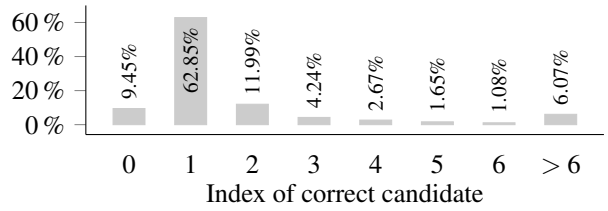


Figure 7: Position of the correct word in the ranked candidate list. Zero means the correct word was not in the candidate list

pact on the results, providing further evidence that the memory operand and its TLB mapping are the keys to the experiment.

7.2 Cache Side Channel

We evaluated the effectiveness of the cache side channel using a microbenchmark application that repeatedly accesses a 4 KB page-aligned array at random indices. We executed the application in an enclave and ran our driver to observe the cache lines of the array accesses.

Figure 6 shows the distribution of the number of cache line candidates returned by the attack over 35,000 accesses with and without applying deterministic filtering (Section 4.3). Both cases use noise filtering (Section 6.4).

With deterministic filtering, we obtained a unique cache line candidate in 87% of the observations, where 99.4% of these unique candidates identified the correct cache line. This level of accuracy is considerable, given that each observations is based on only a single memory access by the victim. The number also reveals the noticable presence of residual noise. As we will show next, this noise degrades the results of our attacks only moderately.

7.3 VC3

We used 20 English books from Project Gutenberg [2] and an English language model with 124,758 unigrams and 912,125 bigrams to evaluate our attack on VC3.

Effectiveness We ran our WordCount application on VC3 inside an enclave for each of the 20 books. We used our driver to perform the attack steps specified in Section 5. The driver produced a log file containing its observations, which we processed as described in Section 5.1.5.

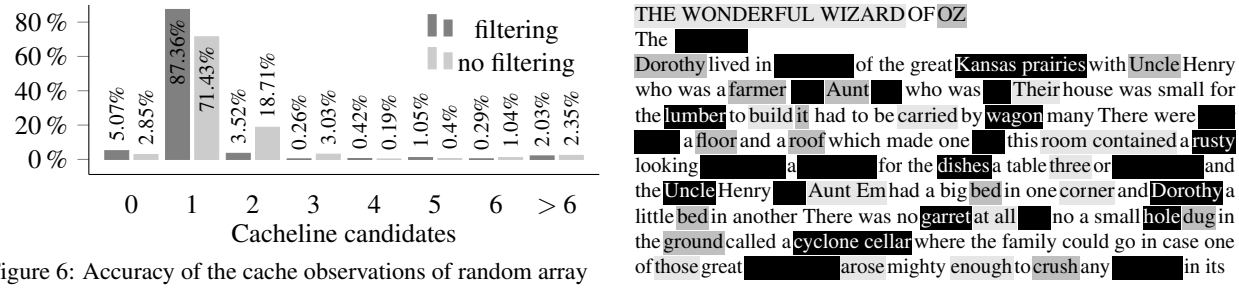


Figure 8: A Sample of the text recovered by the attack: white background: 1st candidate; light-grey background: 2nd candidate; dark-grey background: 3rd candidate; black background: 4th or higher; solid black: word not in candidate list

We evaluated the accuracy of the resulting candidate lists by looking up where each word from the input document appeared in its candidate list. Figure 7 summarizes the results for one book [8] totaling 35,718 words. It shows the position of the correct word in each word's candidate list. For almost two thirds of the words (63%), the first candidate in the list is exactly the word in the document.

Figure 8 displays a sample of the recovered text. While recovery is not perfect, most of the words have been recovered uniquely or nearly uniquely. Overall, the content of the input text is revealed and comprehensible, in spite of our crude lanugage model which lacks explicit grammar rules. A better model is bound to help recover even more of the input.

Performance The end-to-end attacks slow down the applications' execution significantly. Following the approach of [44], our goal is to keep this slowdown at a level that can plausibly be explained by network and scheduling delays and various other cloud and internet glitches.

The runtime for our example book [8] incresases from 0.33 s to 123.5 s when deploying our attack, a 374x overhead similar to the previous attack [44]. The delay can be reduced by extracting only part of the document or by running several mappers and reducers in parallel. Word-length recovery (single stepping 560,128 times) and observing the hash slots (cache side channel) each take slightly less than half of the overhead (57 s and 54.7 s respectively), while 11.6 s are spent handling page faults.

7.4 JPEG

We used 20 images from Wikipedia [5] to evaluate the attack on `libjpeg`. Our test set included some of the images used in an earlier attack on `libjpeg` [44].

Effectiveness Figure 9 shows the result for an image from the earlier attack. The image recovered by our attack shows two expected artifact types. First, loss of detail due to cache-line granular observations. Second, noise,

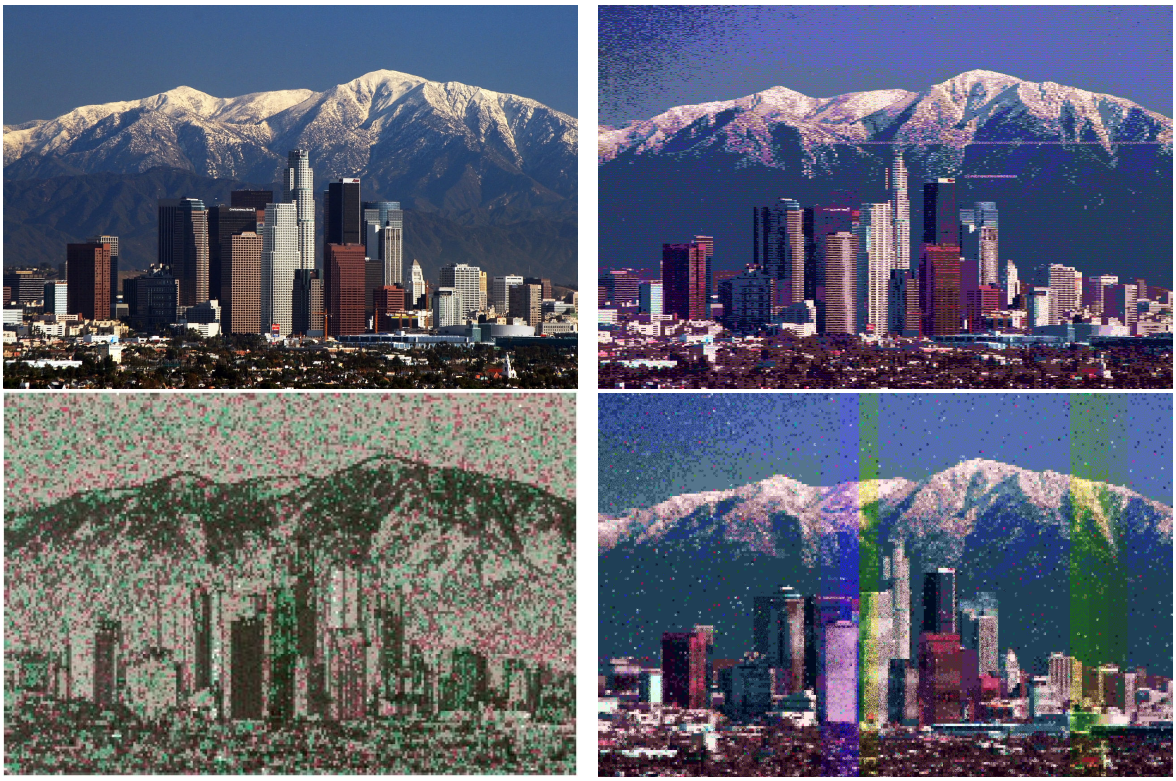


Figure 9: Example of a JPEG image recovered by the attack. top left: the original image; top right: recovered by the full attack; bottom left: recovered through the page fault channel [44, Figure 11]; bottom right: recovered by our attack, sampling 1 to 64.

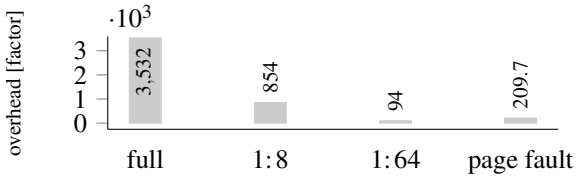


Figure 10: Overhead over baseline of various versions of the attack on libjpeg on the image in Figure 9.

resulting from incorrect cacheline observations. Despite these artifacts, the image recovered by our attack (Figure 9 top right) shows far more detail and looks much more like the original (Figure 9 top left) than the image recovered by the earlier attack (Figure 9 bottom left).[2]

Performance The attack incurs a substantial overhead due to the large number of prime-probe observations (up to three per pixel) and their relatively high cost. Our full attack on the image in Figure 9 incurs a 3,532x overhead (219 s vs. 62 ms), which is substantially higher than the 209.6x-354.9x for the page-fault channel [44, Fig.14].

However, the attacker can easily trade off overhead against accuracy by performing prime-probe observations only for a subset of the application's array lookups. We implemented this sampling strategy by allowing the attacker to specify how many values should be sampled in each 8x8 block. We repeated the attack sampling eight times per block and once per block, corresponding to 1:8 and 1:64 sampling ratios, and summarize the overheads in Figure 10. We show the median overhead for ten measurements. The standard deviation was less than 5% of the mean in all cases. The *page-fault* value is the 209.6x value reported for the previous page-fault channel attack [44, Fig.14].

The two bottom-row images in Figure 9 show clearly that even the image recovered at a 1:64 sampling ratio contains significantly more detail than the image recovered in [44], despite our significantly smaller overhead.

The overheads for the other images in our test set are similar to those of Figure 10, ranging from 2,595x to 3,532x for the full attack and 72x-94x for 1:64 sampling. The attack delays range from less than 10 s to about 4 min for the full attack, and 1.8 s-6 s for 1:64 sampling.

8 Mitigations

Cache side channels have been known for a long time, and a variety of defenses has been designed against them [15, 17, 19, 21, 32, 43, 47], working at the hardware, hypervisor, OS or compiler level. One approach is to partition caches so that the cache assigned to a sensitive

[2]Full resolution and additional images at tudos.org/~mhaehnel/SGX/

application cannot be accessed by a malicious program (e.g., [19, 32]). The other approach is to introduce noise so that a malicious program cannot tell if a cache miss is due to a real or random memory access (e.g., [21, 47]).

None of these defenses appears to be widely used or deployed, possibly due to their cost. In addition, traditional cache side channels have been targeting almost exclusively a small collection of cryptographic algorithms. These have been protected by eliminating all secret-dependent memory accesses from their implementations, thus obviating the need for more general defenses. However, attacks such as those presented in this paper demonstrate that a much broader class of code is potentially subject to cache side-channel attacks when the operating system is the adversary, and general defenses like those listed above may be required.

Shih *et al.* [41] propose a technique called T-SGX to disable side channels based on page-faults and interrupts. T-SGX is a compiler-based approach that automatically wraps computations in Intel TSX transactions. Since TSX [16] aborts transactions upon interrupts and exceptions, T-SGX can use the frequency of such aborts to detect side-channel attacks. T-SGX appears effective, but requires the application source code and incurs a noticeable overhead.

9 Related Work

Untrusted OSs Using feature-rich commodity OSs while removing them from applications' TCB has attracted considerable attention in research and industry. Hardware such as the Trusted Platform Module [3], Intel Trusted Execution Technology [22] or ARM TrustZone [4] as well as software hypervisors have long formed the basis for such systems. Some require applications to be specifically written for the new environment [20], others have the ambitious goal of securing legacy applications [14, 27].

Recently, the goal of protecting user applications in the cloud from the hosting provider's privileged software together with the introduction of Intel SGX have resulted in renewed activity in this area [6, 9, 26, 29, 34, 39].

Xu *et al.* [44] recognized the OS to be significantly more powerful than the traditional unprivileged attacker assumed by most side channel attacks. Their page-fault channel attack extracts complete text documents and outlines of JPEG images from a single run of the victim.

The channels presented in this paper offer much higher spatial and temporal resolution than the page-fault channel. This is significant because it shows that the collection of vulnerable application code is far broader than suggested by the page-fault channel.

Cache side channels Cache-based timing and trace-driven attacks are closely related to our work [11, 36, 37]. They generally assume an attacker with low privilege such as an unprivileged process [37], a virtual machine attacking its neighbors [38] or an attacker measuring server response time across the network [11]. Our attack is trace-driven, as the attacker can observe the victim's memory accesses. Trace-driven attacks typically reveal more fine-grained information than timing attacks. Most trace-based attacks are based on one of two techniques: prime-and-probe [36] and flush-and-reload [24].

The prime-and-probe technique has been used in synchronous [36] and asynchronous attacks [30, 33], targeting the L1 cache [37] and well as the LLC [30, 33]. This line of work typically assumes an unprivileged attacker who has little control over code running between the prime and the probe step. In contrast, control over scheduling and the ability to break into the victim at will let our attacker observe individual memory accesses at high time resolution. Several recent papers study cache side channels for enclaves using techniques that are different from ours and focusing primarily on crypto targets [12, 23, 40]. While preparing the camera-ready version of this paper, we became aware of recent, unpublished work that uses techniques similar to ours to attack crypto code [35].

Recently, the flush-and-reload method has enabled an array of stronger attacks [10, 25, 42, 45, 46]. Flush-and-reload can be used if the attacker and the victim share memory such as read-only code pages. While applicable in traditional cloud scenarios or with memory deduplication, our security model (and SGX) prevent such sharing. Furthermore our attacks are not limited to shared read-only code and data pages.

10 Conclusion

We have described two general techniques that can be combined to build high-resolution side channels for untrusted OSs, overcoming the main limitations of the page-fault channel.

This work shows that a much wider range of application code than suggested by the page-fault channel is subject to side-channel attacks. We demonstrate this with attacks against application code that cannot be attacked using only the page-fault channel. Whereas previous attacks have focused on unmodified legacy code, our main attack successfully targets an application that was designed not to trust the OS. This work highlights the increased importance of side-channels for privileged attacker scenarios. It takes us closer to understanding the full scope of the side-channel problem for untrusted OSs and highlights the need for robust mitigations.

Acknowledgments

We would like to thank our shepherd Nadav Amit and the anonymous reviewers for their valuable input.

References

[1] libjpeg. http://libjpeg.sourceforge.net/.

[2] Project Gutenberg. http://www.gutenberg.org/.

[3] Trusted Platform Module (TPM). http://www.trustedcomputinggroup.org/.

[4] TrustZone. http://www.arm.com/products/processors / technologies / trustzone / index.php.

[5] Wikipedia. http://www.wikipedia.org/.

[6] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP 2013)*, June 2013.

[7] Apache Software Foundation. Hadoop. http://wiki.apache.org/hadoop/, 2011.

[8] L Frank Baum. *The wonderful wizard of Oz*.

[9] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[10] Naomi Benger, Joop van de Pol, Nigel Smart, and Yuval Yarom. "Ooh aah... just a little bit": A small amount of side channel can go a long way. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2014.

[11] Daniel J. Bernstein. Cache-timing attacks on AES. Available at: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2005.

[12] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. https://arxiv.org/abs/1702.07521, February 2017.

[13] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[14] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[15] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, pages 45–60, 2009.

[16] Intel Corporation. Intel 64 and ia-32 architectures software developer's manual.

[17] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting cache side-channel attacks through dynamic software diversity. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS 2015)*, 2015.

[18] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th USENIX Symposium on Operating System Design and Implementation (OSDI'04)*, December 2004.

[19] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization*, 8(4):35:1–35:21, January 2012.

[20] Paul England, Butler Lampson, John Manferdelli, Marcus Peinado, and Bryan Willman. A trusted open platform. *Computer*, 36(7):55–62, 2003.

[21] Adi Fuchs and Ruby B. Lee. Disruptive Prefetching: Impact on Side-Channel Attacks and Cache Designs. In *Proceedings of the 8th ACM International Systems and Storage Conference*, May 2015.

[22] Matthew Gillespie. Intel trusted execution technology: A primer. https://software.intel.com/en-us/articles/intel-trusted-execution-technology-a-primer/.

[23] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache atacks on SGX.

In *2017 European Workshop on Systems Security (EuroSec'17)*.

[24] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – bringing access-based cache attacks on AES to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, pages 490 –505, May 2011.

[25] Berk Gülmezoglu, Mehmet Sinan Inci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. A faster and more realistic flush+reload attack on AES. In *Proceedings of Constructive Side-Channel Analysis and Secure Design COSADE (2015)*, 2015.

[26] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Carlos Rozas, Vinay Phegade, and Juan del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP 2013)*, June 2013.

[27] Owen S. Hofmann, Alan M. Dunn, Sangman Kim, Michael Z. Lee, and Emmett Witchel. InkTag: Secure applications on an untrusted operating system. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[28] Intel Corp. *Software Guard Extensions Programming Reference*, September 2013. Ref. #329298-001 `http://software.intel.com/sites/default/files/329298-001.pdf`.

[29] Intel Corp. *Software Guard Extensions Programming Reference*, October 2014. Ref. #329298-002US `https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf`.

[30] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A shared cache attack that works across cores and defies vm sandboxingand its application to AES. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015.

[31] Prerit Jain, Soham Desai, Seongmin Kim, Ming-Wei Shih, JaeHyuk Lee, Changho Choi, Youjung Shin, Taesoo Kim, Brent B. Kang, and Dongsu Han. Opensgx: An open platform for sgx research. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS 2016)*, 2016.

[32] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. In *USENIX Security Symposium*, 2012.

[33] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015.

[34] Frank Mckeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. Innovative instructions and software model for isolated execution. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP 2013)*, June 2013.

[35] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. `https://arxiv.org/abs/1703.06986`, March 2017.

[36] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Topics in Cryptology–CT-RSA 2006*, pages 1–20. Springer, 2006.

[37] Colin Percival. Cache missing for fun and profit. In *BSDCan 2005*, Ottawa, 2005.

[38] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[39] Felix Schuster, Manuel Costa, Cedric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *36th IEEE Symposium on Security and Privacy*. IEEE Institute of Electrical and Electronics Engineers, May 2015.

[40] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *14th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'17)*, 2017.

[41] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2017.

[42] Joop van de Pol, Nigel Smart, and Yuval Yarom. Just a little bit more. In *CT-RSA*, 2015.

[43] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 494–505, 2007.

[44] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, 2015.

[45] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium*, 2014.

[46] Yinqian Zhang, Ari Juels, Michael Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[47] Yinqian Zhang and Michael K Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

Understanding Security Implications of Using Containers in the Cloud

Byungchul Tak[†], Canturk Isci, Sastry Duri, Nilton Bila, Shripad Nadgowda, James Doran
[†]*Kyungpook National University*, *Daegu, Republic of Korea*
IBM TJ Watson Research Center, *Yorktown Heights, NY USA*

Abstract

Container technology is being adopted as a mainstream platform for IT solutions because of high degree of agility, reusability and portability it offers. However, there are challenges to be addressed for successful adoption. First, it is difficult to establish the full pedigree of images downloaded from public registries. Some might have vulnerabilities introduced unintentionally through rounds of updates by different users. Second, non-conformance to the immutable software deployment policies, such as those promoted by the DevOps principles, introduces vulnerabilities and the loss of control over deployed software. In this study, we investigate containers deployed in a production cloud to derive a set of recommended approaches to address these challenges. Our analysis reveals evidences that (i), images of unresolved pedigree have introduced vulnerabilities to containers belonging to third parties; (ii), updates to live public containers are common, defying the tenet that deployed software is immutable; and (iii), scanning containers or images alone is insufficient to eradicate vulnerabilities from public containers. We advocate for better systems support for tracking image provenance and resolving disruptive changes to containers, and propose practices that container users should adopt to limit the vulnerability of their containers.

1 Introduction

Containers are expanding their adoption in the IT arena rapidly as evidenced by recent launches of IBM Bluemix [20], Amazon ECS [10], Azure Container Service [13] and Google Container Engine [16]. Reasons are plentiful. The motto of '*Build, Ship and Run*', easy reusability of images, easy distribution of code, and simplicity of *pick and run* all align well with the agility, portability, visibility goals of modern software development and DevOps principles [14, 26, 30]. Ultimately the goal is to shorten the release cycles, and thus time-to-market, as much as possible.

However, with mainstream adoption of containers, new challenges emerge. Among them, we focus on the following two that we believe are the most critical. First, the ease of distribution and reuse of containers make it difficult to fully understand the origin and pedigree of images we use. Consider this scenario where two benign development actions can lead to a serious security exposure. A developer builds an image with the password-based authentication enabled and pushes it to an image registry. Another developer, unaware of this, pulls this image and builds a database application on top, where the database application adds a default user ID and a password during its installation. This new image is now pushed back to the registry. As a result of these independent actions we end up with an image of a high-risk security exposure that is ready to be pulled and deployed by many unsuspecting users. Anyone can freely use this image to deploy the same database application in the cloud, and it could be one of yours as well. Unintended vulnerabilities could be introduced this way to an image and can quickly spread in the cloud [8, 19].

A second challenge arises where the expectations from the employed DevOps practices do not match reality with containerized application deployments. Modern DevOps practices advocate an "immutable architecture" model, where all software, system and infrastructure requirements of an application are expressed as code. This gives developers the ability to re-create the infrastructure and applications in a repeatable and agile way. Containers, with their ability to package all system and software requirements, are a key enabler for this immutable architecture model. However, there is a commonly observed mismatch, or *drift*, between the declared architecture and the actually deployed application instances in the cloud. This deviation stems from several factors such as in-place updates (e.g., manual configuration change), dynamic configuration and application updates. Such drift can introduce unexpected exposures and side effects on deployed applications and can go unnoticed for a long

time with only the image-centric validation processes.

In this paper we present real examples to these challenges based on our experiences with an internal, production cloud. We demonstrate an actual case study on the image provenance and its implications. We present a detailed data analysis on the extent of the observed drift in cloud, its root causes and mitigation techniques. We demonstrate the value of automated and systematic scanning of container images and live instances to address these challenges for emerging solutions in this space [1, 9, 11, 12, 17, 18, 21, 29, 31]. We present key insights derived from observing aggregate cloud data on security, provenance and drift. Our analysis shows that drift exists in less than 5% of our scanned containers, it has diverse causes, and in some cases can lead to significant vulnerabilities. Our analysis shows that image-centric security solutions are insufficient, and continuous scanning of images and live containers, coupled with good DevOps practices are required to ensure high level of cloud security.

Overall, our contribution can be summarized as: *(i)* Sharing of our experiences of analyzing the security states of containers and images from a production-level container cloud, *(ii)* Detailed drift analysis to understand to what extent it occurs in the production cloud and what the common characteristics are. Based on the analysis, we describe comprehensive list of causes of drift, *(iii)* Lessons and suggestions of approaches we should take to continue to improve the safety of the container cloud.

2 Image & Container Checking

Our security scanning mechanism is fully integrated into a production-level container cloud used internally. It is automatically triggered upon new image pushes and new container launches. It extracts various features such as list of files with attributes, list of installed packages, OS information, docker inspect, and docker history as presented in [22]. It then feeds the extracted features to compliance and package vulnerability checkers, and persists the output of these checkers into store for aggregate and historical analysis. Images that users push come from various sources. One major source is the public docker hub. Another is the IBM's official container images. Users may also choose to create their own images from scratch and push them to the registry.

Compliance Checking: Compliance rules used in our analysis are based on set of best practices recommended by IBM internally to minimize the chances of compromise. Complete list of rules we use in the scan is described in Table 1. Rules are largely categorized into (i) password restrictions (Rules 2B-D), (ii) file system integrity and (iii) remote access packages (Rules 9A-G). Of particular interest is SSH-related rules - 9A, 9E, 9F and 9G. For us these are considered critical rules because

ID	Rules
1A	Each UID must be used only once.
2B	Maximum password age must be set to 90 days.
2C	Minimum password length must be 8.
2D	Minimum days that must elapse between user-initiated password changes should be 1.
5A,B	RD/WR access of root/.rhosts,.netrc only by root
5D,E	Permission of /usr,/etc must be r-x or more restrictive.
5F	The file /etc/security/opasswd must exist and the permission must be rw——- or more restrictive.
5J	Permission settings of /var for other must be r-x or more restrictive.
5K	Permission of /var/tmp must be rwxrwxrwt.
5L	Permission setting of /var/log for other must be r-x or more restrictive.
5M	Permission check of /var/log/faillog
5N	Permission check of /var/log/tallylog
5S	Permission check of snmpd.conf
6D,E,F	wtmp/faillog/tallylog must file exist.
8O	no_hosts_equiv must be present
9A	SSH server must not have been installed.
9B	Telnet server must not have been installed.
9C	Rsh server must not have been installed.
9D	Ftp server must not have been installed.
9E	SSH server must not be enabled.
9F	SSH password authentication should not be enabled.
9G	All passwords must not be weak.

Table 1: List of home-grown compliance rules.

SSH can often be an easy entry point for an attack. If sshd runs on a container that has any user ID with weak password, which is not uncommon, such container could be cracked even with simple dictionary-based attacks.

Package Vulnerability Checking: Vulnerabilities in software are announced via the National Vulnerability Database (NVD) [24]. Each vulnerability is assigned a unique id known as Common Vulnerability Exposure (CVE) ID [15], and given a score to communicate the impact of the vulnerability. In addition, it also lists specific versions of the products affected by the given vulnerability. Our vulnerability checker uses above information to determine vulnerability status of images and containers. Container images and running instances are scanned periodically to determine their vulnerabilities status. One of the consequences of repeated scanning is an image that has no vulnerabilities in a given scan, but may turn out to be vulnerable later.

3 Image Security: Unsafe Pedigree

The foremost challenge of adopting the container cloud identified earlier is the difficulty with grasping the full history of what updates have been applied to the image to be in current state. This means that the base image you pull may contain unidentified vulnerabilities whether it was crafted or inadvertent. Even worse, multiple series of modifications and re-push by different users, including yours, may jointly create unexpected vulnerabilities. Thus, it is naive to expect that images would stay clean even if users strictly follow best practice guidelines. In this section we make a case for the importance of systematic and automated image scan to deal with such issues. We drive our discussion using one actual scenario we encountered.

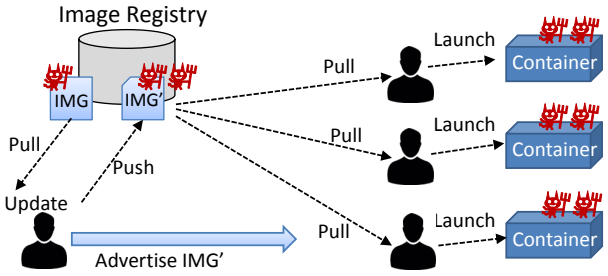


Figure 1: Scenario of vulnerability spread

Case Study: Recently we have come across a puzzling pattern in one of the analytics data. We were looking at the list of about 50 containers that were classified as 'high-risk' that violated SSH-related rules 9A, 9F and 9G. They all had SSH server running, password-authentication enabled and the an ID with weak password. What was most peculiar was that the source image names of all of them had common part, say "*myappsrv*" [1], as if they were all created by one owner. But all of them belonged to different users. How can we explain this phenomenon in which all different users launched containers whose images seemed to have derived from the same source at the same time?

To find an answer we started with searching the Docker Hub for the image that contained "*myappsrv*". We found a candidate, but lacked description. The 'docker inspect' output had the postgres start up commands as the entry point. And, several ports (tcp 22, 5432, 7276, 7286, 9080, 9443) were open. List of packages installed in the image also indicated that it was a postgresql database with SSH server. With further investigation we eventually learned that this image was used in an online course. Students were instructed to pull, customize and launch a container from it.

Figure 1 illustrates the spread process. The instructor pulls the image that already had a postgres server with a default ID of 'postgres' with weak password. Without knowing the existence of this ID, he installs SSH server packages. This image is pushed to the registry and advertised to all the online students. Students pull it and launch containers of their own, resulting in large number of high-risk containers. The instructor was unaware that the original image had an ID with weak password. Also, when installing the SSH server, the intention was to allow only the key-based authentication. In the config file, this line was commented out as this.

```
#PasswordAuthentication yes
```

However, if it is commented out the default behavior of sshd is to enable it. It is easy to be misled to believe that the password authentication is disabled. As a result of all of these, high-risk containers came to life.

[1] Actual name not revealed for a security reason.

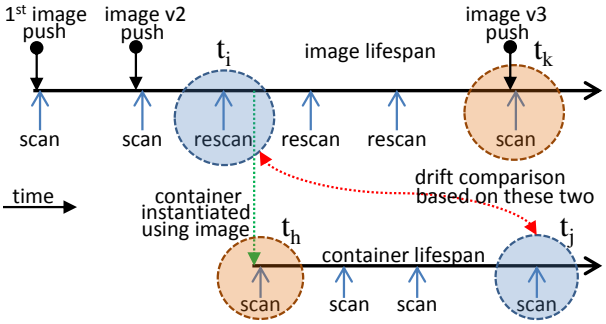


Figure 2: Our definition of drift

Discussion: It is worthwhile making a few points from this case study. First, vulnerabilities can creep in through accumulation of innocuous updates and it is difficult to foresee. Second, we started out with noticing a common pattern in image names across group of vulnerable containers. This exemplifies the advantage of analyzing the aggregate data as a whole which may lead to useful information that eventually reveals the root cause. Based on the observation we make the following statement. *Ensuring the safety of the container cloud should not solely be dependent on users behaving in responsible manner. We must rely on the automated solutions that perform security scans frequently and analyze the data as a whole.*

4 Container Security: Drift Analysis

Here, we analyze the data collected from production-level container cloud, used internally, to understand the drift behavior. The data is collected from two instances of the container cloud operating independently of each other for about two week period in Oct, 2016.

The questions we are interested in are: Does drift exist? If so, how many containers exhibit the drift? Between the compliance and vulnerability, which is the cause of drift? In compliance, which rules in specific are causing the drift? Does drift always increase, or is there a case where it decreases as well and what are they?

4.1 Definition of Drift

We specifically consider the *drift* in terms of the compliance violations and vulnerabilities found. The drift in compliance is defined as the difference between the number of compliance rules violated in a running container and in its corresponding image. Likewise the drift in vulnerability is defined as the difference of the number of vulnerable packages. Other tools (such as Salt [2] and Puppet [3]) may use the *drift* to mean specifically the configuration changes between two states.

Figure 2 illustrates the time aspect of comparison in determining the drift. The upper horizontal arrow indicates the life span of an image. From the moment it is pushed, it is scanned for the compliance and vulnerability. A push of newer version also triggers a new scan.

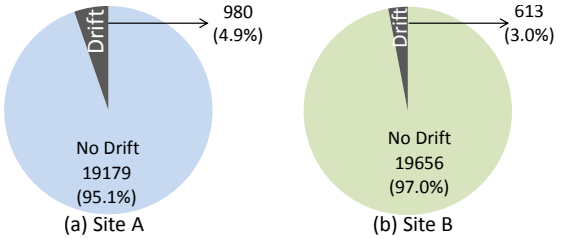


Figure 3: Proportion of containers having drifts.

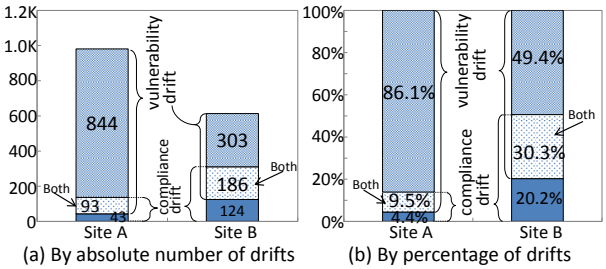


Figure 4: Drift break-down into compliance and vulnerability.

There are scans, labeled as 'rescan', that are not triggered by a push. Rescans are to ensure that results are up-to-date with respect to the updated compliance rules and vulnerable package definitions. The lower horizontal arrow represents the lifespan of a container.

We compare the scan results between t_i and t_j to obtain the drift, denoted as $D(t_i,t_j)$. $D(t_k,t_j)$ is dismissed because t_k might contain new updates that are independent of the container and, thus, making it difficult to identify true divergence of states. Also, $D(t_h,t_j)$ is inappropriate since it may miss $D(t_i,t_h)$ that occurs at the launch time.

4.2 Analysis and Highlights of Findings

We find that drifts do exist in the production containers and the magnitude is less than 5%. As shown in Figure 3, 4.9% and 3.0% of the containers exhibit drifts in both Site A and B, respectively. The existence of drifts, even as small as 5%, is intriguing because ideally the drift is not expected to occur. One harmless cause of the drifts would be the increased number of vulnerable packages in containers not because they actually increased, but because the list of known vulnerable packages grew over time. This raises a question as to how many drifts fall under such category. Also, the site difference of 1.9% seems to be meaningful to deserve a closer look. To find the explanation, we look at the break-down of drift.

Figure 4 is the diagram of drifts broken down into compliance and vulnerability. The existence of compliance drifts tells us that there are other types of drifts than the ones due to the growing definition of vulnerable packages. What is common for both sites is that vulnerability drifts dominates. However, the proportion of vulnerability vs. compliance drifts shows notable difference. Site A has much smaller ratio of compliance drifts (13.9%) than the Site B (50.5%). This may be the indication that the characteristics of the containers from both

		Site A		Site B	
Vulnerability	Increased	913	93.2%	295	48.1%
	Decreased	24	2.4%	194	31.6%
Compliance	Increased	72	7.3%	223	36.4%
	Unchanged	2	0.2%	13	2.1%
	Decreased	62	6.3%	74	12.1%

Table 2: Drift categorization in terms of the direction of changes. Percentage is based on the total drifts per site.

Site A			Site B		
Rule	Count	Pct	Rule	Count	Pct
9A	34	47.2%	1A	134	60.1%
1A	25	34.7%	2B	95	42.6%
9F	24	33.3%	2C	90	40.4%
2B	18	25.0%	2D	50	22.4%
2C	18	25.0%	9A	19	8.5%
2D	6	8.3%	5L	11	4.9%
9G	4	5.6%	9F	8	3.6%
5S	1	1.4%	5S	1	0.4%
			9G	1	0.4%

Table 3: Compliance rules violated in drift cases. Refer to Table 1 for the description of rule codes.

are intrinsically different in regard to compliance rules. Also, it is interesting that the absolute number of vulnerability drifts at Site A is twice as many as that of Site B. Note that it does not necessarily imply that containers at Site B are more secure. This means that the vulnerability status does not change across the instantiation as much irrespective of how secure the images and containers are.

Table 2 provides the break-down of drifts in terms of whether the drift count increases or decreases. One example of a decrease is when the user logs in and manually patches vulnerable packages in the container. According to the Table 2, significant portion (31.6%) of the vulnerability drift at Site B is in the '*Decreased*' category. This contrasts with Site A's number which has only 24 (2.4%). In case of the compliance drift, the proportion of the '*Increased*' category for Site B is much larger than that of Site A. Table 3 explains the cause of the difference. It is because of the high proportion of violations of rule 1A (60.1%) which is twice as large in proportion compared to Site A(34.7%). In addition, password-related rules, 2B-D, rank high in the table for Site B whereas SSH-related rules, 9A and 9F, are towards top of the list for Site A. It is interesting to see that, at Site B, violations of password rules occur more than the violation of SSH rules to the running containers. Similarly, the reverse holds for the Site A. Table 4 also shows the composition of rules that are fixed. We can see that there is a tendency of fixing SSH-related violation within containers at Site B. *Although site differences exist, majority of the drifts are due to the changes of vulnerability status. Also, data shows that 'in-place' updates to the containers, both benign and disruptive, are taking place.*

Focus on SSH rules: In this part we specifically study the drift of SSH related rules among the rules in Table 1. Compliance to the SSH related rules is of particular interest because it is one of the most exploited vulnerabilities [4, 5, 6, 7]. Once compromised, the consequence

Site A			Site B		
Rule	Count	Pct	Rule	Count	Pct
2C	28	45.2%	9A	33	44.6%
9G	26	41.9%	9F	32	43.2%
2B	26	41.9%	9G	30	40.5%
9F	19	30.6%	2D	25	33.8%
9A	8	12.9%	2B	12	16.2%
5B	1	1.6%	2C	11	14.9%
			5S	1	1.4%

Table 4: Non-compliances fixed in drifts. (Rules in Table 1)

Category	Site A		Site B	
No SSH, Password become weak	1	1.3%	1	1.2%
SSH installed	31	39.2%	19	23.5%
SSH installed with weak password	3	3.8%		
Password become weak	1	1.3%		
Sum	**36**	**45.6%**	**20**	**24.7%**
Password become strong	21	26.6%	26	32.1%
Password Auth disabled	13	16.5%	2	2.5%
No SSH, password become strong	1	1.3%		
SSH gets removed	8	10.1%	33	40.7%
Sum	**43**	**54.4%**	**61**	**75.3%**
Total	79	100%	81	100%

Table 5: Classification of SSH-related compliance rule drifts.

could be deadly. But, in many cases this vulnerability is exposed out of neglect, and most of the attacks can be prevented even with small awareness.

Table 5 summarizes the findings related to the SSH rules. It classifies the SSH-related drifts into 8 categories and presents the statistics. Proportion of containers with drifts of SSH rules are about 0.4% for both Site A (79/20K) and Site B (81/20K). The upper half of the table represents drift categories that negatively impacts the SSH vulnerabilities. The lower half shows the drifts that strengthen it. Although magnitude differs, there exist drifts that increase the SSH vulnerabilities in both sites. The risk becomes the highest when all three SSH rules are violated whether it was through manual human actions or automated scripts. Our data do not contain direct information of how these SSH-related drifts happened. But, we strongly suspect that many are due to manual install or password change via SSH login. *Overall, our study suggests that the security scanning of images only is insufficient to eliminate the vulnerabilities. Since security status changes while containers are running, it's critical that containers be scanned periodically.*

4.3 Discussion

Why does disruptive drift happen? While industry thinking coalesces around the belief that containers should be immutable [25], our findings have shown that containers deployed in a cloud drift from their original configuration. Drift occurs for several reasons.

• *Update via Remote Shell Access:* Users of Docker containers are able to login into their containers and execute local commands that alter the state of the containers. Containers offer two shell access modalities: native Docker daemon commands (e.g., *exec*, *attach*) and user installed remote shell servers (e.g., SSH login).

• *Automated Software Update:* Owing to a long history of bug and vulnerability discovery in software long after they ship, software often install with default options to automatically install updates as they become available. As developers build container images they often neglect to changing such defaults.

• *Software configured at runtime:* To aid with usability, popular server applications offer Web admin front ends that allow novice and expert users alike to change their configurations long after they have been deployed.

What can we do about drift? Both the systems and container user communities must work together to realize the promise of an immutable infrastructure. Systems must provide better mechanisms to version and track changes and automate detection of drift from desired container state. Container users must also adopt practices that lead to immutability.

• *Systems support:* Disallowing changes altogether on containers is untenable. Applications, even if stateless, often write cache data or logs to the local file system. First we should track all changes made to containers and give users visibility into these changes [23]. Second, systems must recognize benign changes to the container file systems and memory from undesired changes.

• *Best practices:* Users must adopt practices that contribute to immutable infrastructures. The first step is to discontinue bad habits from the time-sharing era of logging in to manually effect changes. DevOps practices require changes to exist as versioned code that is systemically validated before delivery to production environments. Delivery is the replacement of the live container with a new instance containing versioned code.

Some configurations are bound to the application at run time and cannot be built into the container image. One such example is environment specific variables such as the hostname of a service that the container software depends on. For these configurations, developers must rely on configuration management systems that track changes rather than manually feeding the container with arguments in an ad-hoc manner [27, 28]

5 Conclusion

In this paper, we first established the importance of DevOps as a standard software delivery practice for container-based micro-service architecture. And as an underlying principle DevOps requires security assurance over the pedigree of images along with operational immutability for containers instantiated from these images. To substantiate the extent to which these principles are currently violated, we presented our study on analysis of images and containers in production-level container cloud. We also discussed common characteristics and causes of drifts. Thus, there is an increasing need to have a regulatory protocol and enforcement engine in the platform to curb such non-conformity to ensure security and success of DevOps.

References

[1] Docker Bench for Security. https://github.com/docker/docker-bench-security.

[2] Managing configuration drift with Salt and Snapper. https://www.suse.com/communities/blog/managing-configuration-drift-salt-snapper/.

[3] Maximizing IT Security with Configuration Management. https://puppet.com/resources/whitepaper/maximizing-it-security-configuration-management.

[4] Poorly managed SSH keys pose serious risks for most companies. http://www.computerworld.com/article/2488012/ malware-vulnerabilities/poorly-managed-ssh-keys-pose-serious-risks -for-most-companies.html.

[5] Securing your SSH Server. https://blog.rackspace.com/securing-your-ssh-server.

[6] Security Guide: How to Protect Your Infrastructure Against the Basic Attacker. http://blog.mailgun.com/security-guide-basic-infrastructure-security/.

[7] Security Notice: OpenSSH Passwords. http://authy.com/blog/openssh-passwords-vulnerable/.

[8] Someone said that 30vulnerabilities. https://jpetazzo.github.io/2015/05/27/docker-images-vulnerabilities/.

[9] Understanding Docker Security And Best Practices. https://blog.docker.com/2015/05/understanding-docker-security-and-best-practices/.

[10] AMAZON EC2 CONTAINER SERVICE. https://aws.amazon.com/ecs/.

[11] AMAZON INSPECTOR. https://aws.amazon.com/inspector/.

[12] AQUA. https://www.aquasec.com/.

[13] AZURE CONTAINER SERVICE. https://azure.microsoft.com/en-us/services/container-service/.

[14] CALLANAN, M., AND SPILLANE, A. Devops: Making it easy to do the right thing. *IEEE Software 33*, 3 (May 2016), 53–59.

[15] COMMON VULNERABILITIES AND EXPOSURES. https://cve.mitre.org/.

[16] CONTAINER ENGINE. https://cloud.google.com/container-engine/.

[17] COREOS CLAIR. https://coreos.com/blog/vulnerability-analysis-for-containers.html.

[18] DOCKER SECURITY SCANNING. https://docs.docker.com/docker-cloud/builds/image-scan/.

[19] GUMMARAJU, JAYANTH AND DESIKAN,TARUN AND TURNER, YOSHIO. Over 30% of Official Images in Docker Hub Contain High Priority Security Vulnerabilities. https://www.banyanops.com/blog/analyzing-docker-hub/.

[20] IBM BLUEMIX. https://console.ng.bluemix.net/.

[21] IBM VULNERABILITY ADVISOR. https://www.ibm.com/blogs/bluemix/tag/ibm-vulnerability-advisor/.

[22] KOLLER, R., ISCI, C., SUNEJA, S., AND DE LARA, E. Unified monitoring and analytics in the cloud. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)* (Santa Clara, CA, 2015), USENIX Association.

[23] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2005), PLDI '05, ACM, pp. 190–200.

[24] NATIONAL VULNERABILITY DATABASE. https://nvd.nist.gov/.

[25] RAFAEL BENEVIDES. 10 things to avoid in docker containers. https://developers.redhat.com/blog/2016/02/24/10-things-to-avoid-in-docker-containers/.

[26] ROCHE, J. Adopting devops practices in quality assurance. *Commun. ACM 56*, 11 (Nov. 2013), 38–43.

[27] SHERMAN, A., LISIECKI, P. A., BERKHEIMER, A., AND WEIN, J. Acms: The akamai configuration management system. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2* (Berkeley, CA, USA, 2005), NSDI'05, USENIX Association, pp. 245–258.

[28] TANG, C., KOOBURAT, T., VENKATACHALAM, P., CHANDER, A., WEN, Z., NARAYANAN, A., DOWELL, P., AND KARL, R. Holistic configuration management at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 328–343.

[29] TWISTLOCK. `https://www.twistlock.com/`.

[30] WETTINGER, J., ANDRIKOPOULOS, V., AND LEYMANN, F. Automated capturing and systematic usage of devops knowledge for cloud applications. In *2015 IEEE International Conference on Cloud Engineering* (March 2015), pp. 60–65.

[31] WINKEL, S. Security Assurance of Docker Container. In *SANS Institute Whitepaper* (October 2016). https://www.sans.org/reading-room/whitepapers/assurance/security-assurance-docker-containers-37432.

Memshare: a Dynamic Multi-tenant Key-value Cache

Asaf Cidon*, Daniel Rushton†, Stephen M. Rumble‡, Ryan Stutsman†
*Stanford University, †University of Utah, ‡Google Inc.

Abstract

Web application performance heavily relies on the hit rate of DRAM key-value caches. Current DRAM caches statically partition memory across applications that share the cache. This results in under utilization and limits cache hit rates. We present Memshare, a DRAM key-value cache that dynamically manages memory across applications. Memshare provides a resource sharing model that guarantees reserved memory to different applications while dynamically pooling and sharing the remaining memory to optimize overall hit rate.

Key-value caches are typically memory capacity bound, which leaves cache server CPU and memory bandwidth idle. Memshare leverages these resources with a log-structured design that allows it to provide better hit rates than conventional caches by dynamically re-partitioning memory among applications. We implemented Memshare and ran it on a week-long trace from a commercial memcached provider. Memshare increases the combined hit rate of the applications in the trace from 84.7% to 90.8%, and it reduces the total number of misses by 39.7% without significantly affecting cache throughput or latency. Even for single-tenant applications, Memshare increases the average hit rate of the state-of-the-art key-value cache by an additional 2.7%.

1 Introduction

DRAM key-value caches are essential for reducing application latency and absorbing massive database request loads in web applications. For example, Facebook has dozens of applications that access hundreds of terabytes of data stored in memcached [24] in-memory caches [41]. Smaller companies use outsourced multi-tenant in-memory caches to cost-effectively boost SQL database performance.

High access rates and slow backend database performance mean reducing cache miss rates directly translates to end-to-end application performance. For example, one Facebook memcached pool achieves a 98.2% hit rate [9]. With an average cache latency of 100 µs and MySQL access times of 10 ms, increasing the hit rate by 1% reduces latency by 36% (from 278 µs to 179 µs) and reduces database read load by 2.3×.

Today, operators statically divide memory across applications. For example, Facebook, which manages its own data centers and cache clusters [9, 39], has an engineer that is tasked to manually partition machines into separate cache pools for isolation. Similarly, Memcachier [4, 18], a cache-as-a-service for hundreds of tenants, requires customers to purchase a fixed amount of memory.

Static partitioning is inefficient, especially under changing application loads; some applications habitually under utilize their memory while others are short of resources. Worse, it is difficult for cache operators to decide how much memory should be allocated to each application. This manual partitioning requires constant tuning over time. Ideally, a web cache should automatically learn and assign the optimal memory partitions for each application based on their changing working sets; if an application needs a short term boost in cache capacity, it should be able to borrow memory from one that needs it less, without any human intervention.

To this end, we designed *Memshare*, a multi-tenant DRAM cache that improves cache hit rates by automatically sharing pooled and idle memory resources while providing performance isolation guarantees. To facilitate dynamic partitioning of memory among applications, Memshare stores each application's items in a segmented in-memory log. Memshare uses an *arbiter* to dynamically decide which applications require more memory and which applications are over-provisioned, and it uses a *cleaner* to evict items based on their rank and to compact memory to eliminate fragmentation.

This paper makes two main contributions:

1. Memshare is the first multi-tenant web memory cache that optimally shares memory across applications to maximize hit rates, while providing isolation guarantees. Memshare does this with novel dynamic and automatic profiling and adaptive memory reallocation that boost overall hit rate.
2. Memshare uniquely enforces isolation through a log-structured design with application-aware cleaning that enables fungibility of memory among applications that have items of different sizes. Due to its memory-efficient design, Memshare achieves significantly higher hit rates than the state-of-the-art memory cache, both in multi-tenant environments and in single-tenant environments.

In Memshare, each application specifies a minimum amount of *reserved* memory; the remaining *pooled* memory is used flexibly to maximize hit rate. Inspired by

Cliffhanger [19], Memshare optimizes hit rates by estimating hit rate gradients; it extends this approach to track a gradient for each application, and it awards memory to the applications that can benefit the most from it. This enables cache providers to increase hit rates with fewer memory resources while insulating individual applications from slowdowns due to sharing. Even when all memory is reserved for specific applications, Memshare can increase overall system efficiency without affecting performance isolation by allowing idle memory to be reused between applications. Memshare also lets each application specify its own eviction policy (e.g., LRU, LFU, Segmented LRU) as a *ranking function* [11]. For example, to implement LRU, items are ranked based on the timestamp of their last access; to implement LFU, items are ranked based on their access frequency.

Existing memory caches cannot support these properties; they typically use a slab allocator [3, 18, 19], where items of different sizes are assigned to slab classes and eviction is done independently on a class-by-class basis. This limits their ability to reassign memory between different applications and between items of different sizes.

Memshare replaces slab allocation with a new log-structured allocator that makes memory fungible between items of different sizes and applications. The drawback of the log-structured allocator is that it continuously repacks memory contents to reassign memory, which increases CPU and memory bandwidth use. However, increasing hit rates in exchange for higher CPU and memory bandwidth use is attractive, since key-value caches are typically memory capacity bound and not CPU bound. In a week-long trace from Memcachier, cache inserts induce less than 0.0001% memory bandwidth utilization and similarly negligible CPU overhead. CPU and memory bandwidth should be viewed as under utilized resources that can be used to increase the cache efficiency, which motivates the log-structured approach for memory caches.

Nathan Bronson from the data infrastructure team at Facebook echoes this observation: "Memcached shares a RAM-heavy server configuration with other services that have more demanding CPU requirements, so in practice memcached is never CPU-bound in our data centers. Increasing CPU to improve the hit rate would be a good trade off." [16]. Even under high CPU load, Memshare's cleaner can dynamically shed load by giving up eviction policy accuracy, but, in practice, it strongly enforces global eviction policies like LRU with minimal CPU load.

We implement Memshare and analyze its performance by running a week-long trace from Memcachier, a multi-tenant memcached service [18]. We show that Memshare adds 6.1% to the overall cache hit rate compared to memcached. We demonstrate that Memshare's added overheads do not affect client-observed performance for real workloads, since CPU and memory bandwidth are significantly under utilized. Our experiments show that Memshare achieves its superior hit rates and consumes less than 10 MB/s of memory bandwidth, even under aggressive settings. This represents only about 0.01% of the memory bandwidth of a single CPU socket. We demonstrate that in the case of a single-tenant application running in the cache, Memshare increases the number of hits by an extra 2.37% compared to Cliffhanger [19], the state-of-the-art single-tenant cache. To the best of our knowledge, Memshare achieves significantly higher average hit rates than any other memory cache both for multi-tenant and single-tenant workloads.

2 Motivation

DRAM key-value caches are an essential part of web application infrastructure. Facebook, Twitter, Dropbox, and Box maintain clusters of thousands of dedicated servers that run web caches like memcached [24] that serve a wide variety of real-time and batch applications. Smaller companies use caching-as-a-service providers such as ElastiCache [1], Redis Labs [5] and Memcachier [4]. These multi-tenant cache providers may split a single server's memory among dozens or hundreds of applications.

Today, cache providers partition memory statically across multiple applications. For example, Facebook, which manages its own cache clusters, partitions applications among a handful of pools [9, 39]. Each pool is a cluster of memcached servers that cache items with similar QoS needs. Choosing which applications belong in each pool is done manually. Caching-as-a-service providers like Memcachier [4, 18] let customers purchase a certain amount of memory. Each application is statically allocated memory on several servers, and these servers maintain a separate eviction queue for each application.

2.1 Partitioned vs Pooled

We compare two different resource sharing schemes with memcached using simulation[1]: the static partitioning used by Memcachier, and a greedy pooled memory policy, both using memcached's slab allocator with LRU. In the static partitioning, we run applications just as they run in our commercial Memcachier trace; each is given isolated access to the same amount of memory it had in the trace. In the pooled policy, applications share all memory, and their items share eviction queues. An incoming item from any application evicts items from the tail of the shared per-class eviction queues (§2.2), which are oblivious to which application the items belong to. We use a motivating example of three different applications (3, 5 and 7) selected from a week-long trace of memcached traffic running on Memcachier. These applications suffer from bursts of requests, so they clearly demonstrate the trade offs between the partitioned and pooled memory policies.

[1] Source available at `http://github.com/utah-scs/lsm-sim/`

App	Hit Rate Partitioned	Pooled
3	97.6%	96.6%
5	98.8%	99.1%
7	30.1%	39.2%
Combined	87.8%	88.8%

Table 1: Average hit rate of Memcachier's partitioned and pooled policy over a week.

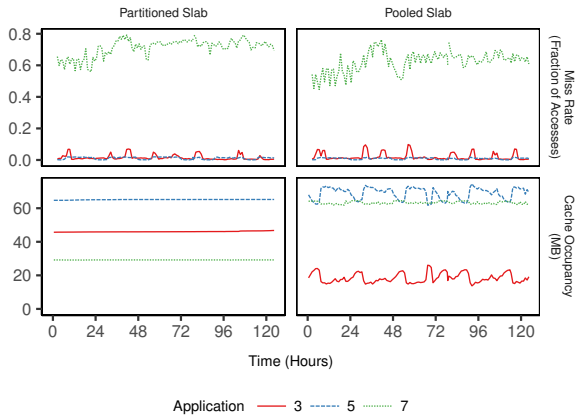


Figure 1: Miss rate and cache occupancy of Memcachier's partitioned and pooled policies over time.

Table 1 shows the average hit rates over a week of the three applications in both configurations. Figure 1 depicts the average miss rate and cache occupancy over the week. The pooled policy gives a superior overall hit rate, but application 3's hit rate drops 1%. This would result in 42% higher database load and increased latencies for that application. The figure also shows that the pooled scheme significantly changes the allocation between the applications; application 3 loses about half its memory, while application 7 doubles its share.

2.2 Slab Allocation Limits Multi-tenancy

Ideally, a multi-tenant eviction policy should combine the best of partitioned and pooled resource sharing. It should provide performance isolation; it should also allow applications to claim unused memory resources when appropriate, so that an application that has a burst of requests can temporarily acquire resources. This raises two requirements for the policy. First, it must be able to dynamically arbiter which applications can best benefit from additional memory and which applications will suffer the least when losing memory. Second, it needs to be able to dynamically reallocate memory across applications.

Unfortunately, allocators like memcached's slab allocator greatly limit the ability to move memory between applications, since items of different sizes are partitioned in their own slabs. The following example illustrates the problem. Imagine moving 4 KB of memory from application 1 to application 3. In the trace, the median item size for application 1 and 3 are 56 B and 576 B, respectively. In Memcachier, each 1 MB slab of memory is assigned a *size class*; the slab is divided into fixed sized chunks according to its class. Classes are in units of 64×2^i up to 1 MB (i.e. 64 B, 128 B, ..., 1 MB). Each item is stored in the smallest class that can contain the item. Therefore, items of 56 B are stored in a 1 MB slab of 64 B chunks, and 576 B are stored in a 1 MB slab of 1 KB chunks.

There are two problems with moving memory across applications in a slab allocator. First, even if only a small amount needs to be moved (4 KB), memory can only be moved in 1 MB units. So, application 1 would have to evict 1 MB full of small items, some of which may be hot; memcached tracks LRU rank via an explicit list, which doesn't relate to how items are physically grouped within slabs. Second, the newly reallocated 1 MB could only be used for a single item size. So, application 3 could only use it for items of size 256-512 B or 512-1024 B. If it needed memory for items of both sizes, it would need application 1 to evict a second slab. Ideally, the cache would only evict the bottom ranked items from application 1, based on application 1's eviction policy, which have a total size of 4 KB. This problem occurs even when assigning memory between different object sizes *within the same application*.

This motivates a new design for a multi-tenant cache memory allocator that can dynamically move variable amounts of memory among applications (and among different object sizes of the same application) while preserving applications' eviction policy and priorities.

3 Design

Memshare is a lookaside cache server that supports the memcached API. Unlike previous key-value caches, Memshare stores items of varying sizes and applications physically together in memory, and uses a cleaner running in the background to remove dead items. When the cache is full, it decides which items to evict based on the items' eviction priorities and how effectively each application uses its share of the cache.

Memshare is split into two key components. First, Memshare's arbiter must determine how much memory should be assigned to each application (its *targetMem*). Second, Memshare's cleaner implements these assignments by prioritizing eviction from applications that are using too much cache space.

3.1 The Cleaner and Arbiter

Memshare's in-memory cleaner fluidly reallocates memory among applications. The cleaner finds and evicts the least useful items for any application from anywhere in memory, and it coalesces the resulting free space for newly written items. This coalescing also provides fast allocation and high memory utilization.

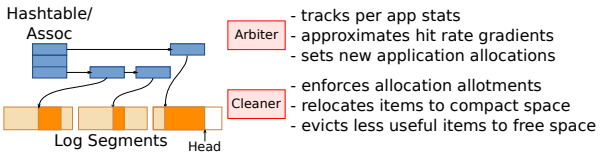


Figure 2: The Memshare design. Incoming items are allocated from the head of a segmented in-memory log. The hash table maps keys to their location in the log. The arbiter monitors operations and sets allocation policy. The cleaner evicts items according to the arbiter's policy and compacts free space.

All items in Memshare are stored in a segmented in-memory log (Figure 2). New items are allocated contiguously from the same active *head* segment, which starts empty and fills front-to-back. Once an item has been appended to the log, the hash table entry for its key is pointed to its new location in the log. Unlike slab allocator systems like memcached, Memshare's segments store items of all sizes from all applications; they are all freely intermixed. By default, segments are 1 MB; when the head segment is full, an empty "free" segment is chosen as head. This accommodates the largest items accepted by memcached and limits internal fragmentation.

When the system is running low on free segments ($< 1\%$ of total DRAM), it begins to run the cleaner in the background, in parallel with handling normal requests. The cleaner frees space in two steps. First, it evicts items that belong to an application that is using too much cache memory. Second, it compacts free space together into whole free segments by moving items in memory. Keeping a small pool of free segments allows the system to tolerate bursts of writes without blocking on cleaning.

Memshare relies on its arbiter to choose which items the cleaner should prefer for eviction. To this end we define the *need* of each application as its need for memory:

$$need(app) = \frac{targetMem(app)}{actualMem(app)}$$

Where *actualMem* is the actual number of bytes currently storing items belonging to the application, and *targetMem* is the number of bytes that the application is supposed to be allocated. In the case of partitioned resource allocation *targetMem* is constant. If the need of an application is above 1, it means it needs to be allocated more memory. Similarly, if the need is below 1, it is consuming more memory than it should. The arbiter ranks applications by their *need* for memory; the cleaner prefers to clean from segments that contain more data from applications that have the lowest need. Items in a segment being cleaned are considered one-by-one; some are saved and others are evicted.

Cleaning works in "passes". Each pass takes n distinct segments and outputs at most $n-1$ new segments, freeing up at least one empty segment. This is done by writing back the most essential items into the $n-1$ output segments. The writing is contiguous so free space, caused by obsolete items that were overwritten, is also eliminated. n is a system parameter that is discussed in Section 6. Note that multiple passes can run in parallel.

Algorithm 1 Memory relocation

```
1:  function CLEANMEMORY(segments, n)
2:      relocated = 0
3:      residual = (n - 1) · segmentSize
4:      while segments not empty do
5:          app = arbiter.maxNeed()
6:          item = maxRank(segments, app)
7:          segments.remove(item)
8:          if item.size ≤ residual - relocated then
9:              relocate(item)
10:             relocated = relocated + item.size
11:             app.actualMem = app.actualMem + item.size
12:         else
13:             break
14:         end if
15:     end while
16: end function
```

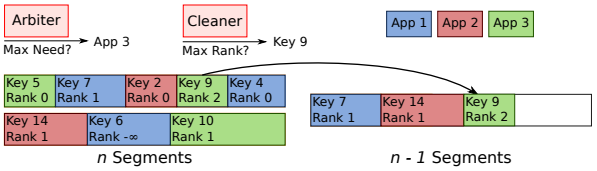


Figure 3: Memshare relocates items from n segments to $n-1$ segments. The arbiter first chooses the application with the highest need, and the cleaner relocates the item with the highest rank among the items of that application.

In each pass, Memshare selects a fraction of the segments for cleaning randomly and a fraction based on which segments have the most data from applications with the lowest need. Random selection helps to avoid pathologies. For example, if segments were only chosen based on application need, some applications might be able to remain over provisioned indefinitely so long as there are worse offenders. Based on experience with the Memcachier traces, choosing half of the segments randomly avoided pathologies while tightly enforcing arbiter policies.

Once a set of segments is selected for cleaning, the cleaner sorts the items in the segments by rank to determine which items should be preserved. Figure 3 and Algorithm 1 show how this is done in a single cleaning pass. *segments* is a list of all the items from the segments being cleaned in the pass. In order to choose which item to relocate next, the cleaner first determines the application that has the highest need (*maxNeed*). Among the items in the segments that belong to that application, the cleaner then chooses the item with the highest rank (*maxRank*, e.g. LRU-rank). It relocates the item by copying it and updating its entry in the hash table. After the item is relocated, the need for that application is recalculated. The

process is repeated until the $n-1$ segments are full or all items are relocated. The remaining items are evicted by dropping them from the hash table, and the need for the applications' whose items were evicted is adjusted.

Memshare can use any generic ranking function on items to prioritize them for eviction; in fact, it can be determined by the application. Memshare supports any ranking function *rank*(t, f), that is based on the timestamp t of the last access of each item and f the number of times it has been accessed. For example, to implement LRU, the ranking function is *rank*$(t) = t$; that is, it is the item's last access timestamp. LFU is just the number of accesses to an item: *rank*$(f) = f$. Segmented LRU can be implemented as a combination of the timestamp of the last access of the item and the number of times it has been accessed. Throughout the paper, when evaluating the hit rate of different caches, we use LRU as the default eviction policy.

A key idea behind Memshare is that memory partitioning is enforced by the decision of which items to clean, while any application can write at any time to the cache. Consider the case where Memshare is configured for a static partitioning among applications, and one application continuously writes new items to the cache while other applications do not. Allocations are static, so *targetMem* will remain constant. As the first application inserts new items, its *actualMem* will increase until its need drops below the need of the other applications. When the memory fills and cleaning starts, the arbiter will choose to clean data from the application that has the lowest need and will begin to evict its data. If there are other active applications competing for memory, this application's *actualMem* will drop, and its need will increase.

3.2 Balancing Eviction Accuracy and Cleaning

The cost of running Memshare is determined by a trade off between the accuracy of the eviction policy, determined by the parameter n and the rate of updates to the cache. The higher the rate of updates, the faster the cleaner must free up memory to keep up. Section 6.1 evaluates this cost and finds for the trace the cleaning cost is less than 0.01% utilization for a single CPU socket. Even so, the cleaner can be made faster and cheaper by decreasing n; decreasing n reduces the amount of the data the cleaner will rewrite to reclaim a segment worth of free space. This also results in the eviction of items that are ranked higher by their respective applications, so the accuracy of the eviction policy decreases. In our design, n can be dynamically adjusted based on the rate of updates to the cache. Web cache workloads typically have a low update rate (less than 3%) [39].

The last of the $n-1$ segments produced by the cleaning pass may be less than full when there are many dead items in the original n segments. The new $n-1$ segments are sorted based on need and rank, so one optimization is to evict the items in last segment if its utilization is low ($< 50\%$) since it contains low rank and need items.

4 Memshare's Sharing Model

Memshare allows the operator to fix a reserved amount of memory for each application. The rest of the cache's memory is pooled and dynamically assigned to the applications whose hit rates would benefit the most from it. Each application's reserved memory we call *reservedMem*; the remaining memory on the server is *pooledMem*, shared among the different applications. At each point in time, Memshare has a target amount of memory it is trying to allocate to each application, *targetMem*. In the case of statically partitioned memory, *pooledMem* is zero, and *targetMem* is always equal to *reservedMem* for each application.

targetMem defines an application's fair share. The resource allocation policy needs to ensure that each application's *targetMem* does not drop below its *reservedMem*, and that the remaining *pooledMem* is distributed among each application in a way that maximizes some performance goal such as the maximum overall hit rate.

To maximize the overall hit rate among the applications, each application's hit rate curve can be estimated; this curve indicates the hit rate the application would achieve for a given amount of memory. Given applications' hit rate curves, memory can be reallocated to applications whose hit rate would benefit the most. However, estimating hit rate curves for each application in a web cache can be expensive and inaccurate [18, 19].

Instead, Memshare estimates local hit rate curve gradients with *shadow queues*. A shadow queue is an extension of the cache that only stores item keys and not item values. Each application has its own shadow queue. Items are evicted from the cache into the shadow queue. For example, imagine an application has 10,000 items stored in the cache, and it has a shadow queue that stores the keys of 1,000 more items. If a request misses the cache and hits in the application's shadow queue, it means that if the application had been allocated space for another 1,000 items, the request would have been a hit. The shadow queue hit rate gives a local approximation of an application's hit rate curve gradient [19]. The application with the highest rate of hits in its shadow queue would provide the highest number of hits if its memory was incrementally increased.

Algorithm 2 shows how *targetMem* is set. Each application is initially given a portion of *pooledMem*. For each cache request that is a miss, the application's shadow queue is checked. If key is present in the shadow queue, that application is assigned a *credit* representing the right to use to a small chunk (e.g., 64 KB) of the pooled memory. Each assigned credit is taken from another application at random (*pickRandom* above). The cleaner uses

Algorithm 2 Pooled memory: set target memory

```
1:  function SETTARGET(request, application)
2:      if request ∉ cache AND
            request ∈ application.shadowQueue then
3:          candidateApps = {}
4:          for app ∈ appList do
5:              if app.pooledMem ≥ credit then
6:                  candidateApps = candidateApps ∪ {app}
7:              end if
8:          end for
9:          pick = pickRandom(candidateApps)
10:         application.pooledMem =
              application.pooledMem + credit
11:         pick.pooledMem = pick.pooledMem - credit
12:     end if
13:     for app ∈ appList do
14:         app.targetMem =
              app.reservedMem + app.pooledMem
15:     end for
16: end function
```

	Hit Rate	
App	Partitioned	Memshare 50%
3	97.6%	99.4%
5	98.8%	98.8%
7	30.1%	34.5%
Combined	87.8%	89.2%

Table 2: Average hit rate of Memshare with 50% reserved memory compared to the partitioned policy.

Reserved Memory	Total Hit Rate
0%	89.4%
25%	89.4%
50%	89.2%
75%	89.0%
100%	88.8%

Table 3: Comparison of Memshare's total hit rate with different amounts of reserved memory for applications 3, 5, and 7.

targetMem to choose which applications to evict items from. *appList* is a list of all applications in the cache and *cache* is a list of all items in the cache.

Table 2 compares Memshare with the statically partitioned Memcachier scheme. For Memshare, each application is configured to use 50% of the memory that was allocated to it in the original trace as reserved memory with the rest as pooled memory. Memshare delivers equal or better hit rates both application-by-application and overall. Even with 50% of memory reserved, Memshare also achieves a higher overall hit rate (89.2%) than the greedy pooled memory scheme (88.8%, Table 1).

Table 3 and Figure 4 further explore the trade off between overall hit rate and per-application hit rates as we vary the percentage of memory that is held reserved. The figure shows that with more memory held reserved, reallocation between applications dampens. In addition, the figure shows Memshare's cleaner enforces the reserved memory allocation for each application: applications never fall below their reservations. The figure also shows how Memshare's memory allocation reacts to the rate of shadow queue hits. In the far left graphs, when the cache has no reserved memory, Memshare allocates pooled memory to the applications that have a high shadow queue hit rate. As Memshare allocates more memory to the bursty application, its shadow queue hit rate tempers. In the far right graphs, when the cache is fully reserved, Memshare cannot allocate any additional memory to the bursty applications; therefore, the shadow queue hit rate remains high.

App	Credit Size	Hit Rate	Credit Size	Hit Rate
3	64 KB	99.4%	64 KB	99.5%
5	128 KB	98.5%	64 KB	98.6%
7	192 KB	33.4%	64 KB	32.3%

Table 4: Assigning different credit sizes to each application allows cache operators to prioritize among applications.

Finally, Table 2 and 3 break down how much of Memshare's hit rate improvements come from its allocator and how much come from its sharing model. With 100% reserved memory, Memshare is equivalent to static partitioning, but it achieves a 88.8% hit rate compared to 87.8% for memcached: a 1% gain strictly due to the allocator. Going from 100% reserved memory to 0% shows a 0.6% gain. This shows that about 38% of Memshare's gains are from memory sharing. Note that effective sharing also requires log-structured allocation.

4.1 Allocation Priority

Cache providers may want to guarantee that when certain applications have bursts of requests, they would get a higher priority than other applications. In order to accommodate this requirement, Memshare enables cache operators to assign different shadow queue credit sizes to different applications. This guarantees that if a certain application has a higher credit size than other applications, when it requires a larger amount of memory due to a burst of activity, it will be able to expand its memory footprint faster than other applications.

Table 4 demonstrates how assigning different weights to different applications affects their overall hit rate. In this example, application 7 achieves a higher relative hit rate, since it receives larger credits in the case of a shadow queue hit.

4.2 Increasing Efficiency for Reserved Memory

Pooled memory works for environments like Facebook's where multiple cooperative applications use a shared caching layer, and the operator wants to provide the best overall performance while providing minimum guarantees to applications. However, in some environments, applications are inherently selfish and would like to maximize their reserved memory, but the cache operator still

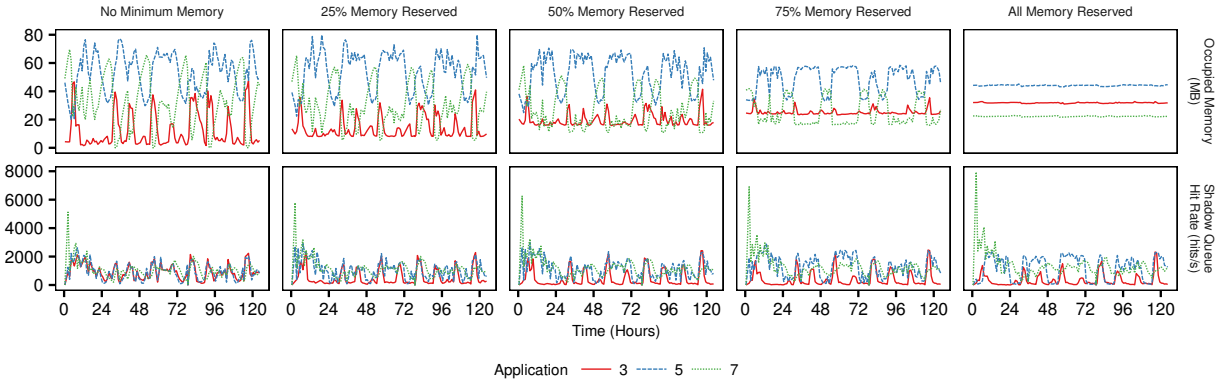


Figure 4: Comparison of Memshare's memory consumption and the rate of shadow queue hits with different amounts of memory reserved for applications 3, 5 and 7. Memshare assigns more pooled memory to applications with a high shadow queue hit rate.

Algorithm 3 Idle tax: set target memory

```
1:  function SETTARGET(app, taxRate, idleTime)
2:      idleMem = 0
3:      for item ∈ app do
4:          if item.timestamp < currentTime - idleTime then
5:              idleMem += item.size
6:          end if
7:      end for
8:      activeFraction = 1 − idleMem / app.actualMem
9:      τ = (1 − activeFraction · taxRate) / (1 − taxRate)
10:     app.targetMem = app.reservedMem / τ
11: end function
```

has an incentive to optimize for effective memory utilization. If applications are "sitting on" their underutilized reserved memory, their resources can be reassigned without negatively impacting their performance.

To help with this, Memshare also supports an idle memory tax that allows memory that has not been accessed for a period to be reassigned. Memshare implements the tax with one small change in how the arbiter sets each application's *targetMem*. Algorithm 3 shows how the arbiter computes *targetMem* for each application when the tax is enabled; $taxRate \in [0, 1]$ determines what fraction of an application's memory can be reassigned if it is idle. If *taxRate* is 1, all of the application's idle memory can be reassigned (and its *targetMem* will be 0). If *taxRate* is 0, the idle tax cache policy is identical to partitioned allocation. Idle memory is any memory that has not been accessed more recently than *idleTime* ago. The arbiter tracks what fraction of each application's memory is idle, and it sets *targetMem* based on the tax rate and the idle fraction for the application.

In this algorithm, *targetMem* cannot be greater than *reservedMem*. If multiple applications have no idle memory and are competing for additional memory, it will be allocated to them in proportion to their *reservedMem*. For example, if two applications with a *targetMem* of 5 MB and 10 MB respectively are contending for 10 MB, the 10 MB will be split in a 1:2 ratio (3.3 MB and 6.7 MB).

App	Hit Rate Memcachier Partitioned	Hit Rate Idle Tax
3	97.6%	99.4%
5	98.8%	98.6%
7	30.1%	31.3%
Combined	87.8%	88.8%

Table 5: Average hit rate of Memshare's idle tax policy.

Table 5 depicts the hit rate Memshare's idle tax algorithm using a tax rate of 50% and a 5 hour idle time. In the three application example, the overall hit rate is increased, because the idle tax cache policy favors items that have been accessed recently. Application 5's hit rate decreases slightly because some of its idle items were accessed after more than 5 hours.

5 Implementation

Memshare consists of three major modules written in C++ on top of memcached 1.4.24: the log, the arbiter and the cleaner. Memshare reuses most of memcached's units without change including its hash table, basic transport, dispatch, and request processing.

5.1 The Log

The log replaces memcached's slab allocator. It provides a basic `alloc` and `free` interface. On allocation, it returns a pointer to the requested number of bytes from the current "head" segment. If the request is too big to fit in the head segment, the log selects an empty segment as the new head and allocates from it.

Allocation of space for new items and the change of a head segment are protected by a spin lock. Contention is not a concern since both operations are inexpensive:

allocation increments an offset in the head segment and changing a head segment requires popping a new segment from a free list. If there were no free segments, threads would block waiting for the cleaner to add new segments to the free list. In practice, the free list is never empty (we describe the reason below).

5.2 The Arbiter

The arbiter tracks two key attributes for each application: the amount of space it occupies and its shadow LRU queue of recently evicted items. The SET request handler forwards each successful SET to the arbiter so the per-application bytes-in-use count can be increased. On evictions during cleaning passes, the arbiter decreases the per-application bytes-in-use count and inserts the evicted items' into the application's shadow queue. In practice, the shadow queue only stores the 64-bit hash of each key and the length of each item that it contains, which makes it small and efficient. Hash collisions are almost non-existent and do no harm; they simply result in slight over-counting of shadow queue hits.

5.3 The Cleaner

The cleaner always tries to keep some free memory available. By default, when less than 1% of memory is free the cleaner begins cleaning. It stops when at least 1% is free again. If the cleaner falls behind the rate at which service threads perform inserts, then it starts new threads and cleans in parallel. The cleaner can clean more aggressively, by reducing the number of segments for cleaning (n) or freeing up more segments in each cleaning pass. This trades eviction policy accuracy for reduced CPU load and memory bandwidth.

Cleaning passes must synchronize with each other and with normal request processing. A spin lock protects the list of full segments and the list of empty segments. They are both manipulated briefly at the start and end of each cleaning pass to choose segments to clean and to acquire or release free segments. In addition, the cleaner uses Memcached's fine-grained bucket locks to synchronize hash table access. The cleaner accesses the hash table to determine item liveness, to evict items, and to update item locations when they are relocated.

The arbiter's per-app bytes-in-use counts and shadow queues are protected by a spin lock, since they must be changed in response to evictions. Each cleaner pass aggregates the total number of bytes evicted from each application and it installs the change with a single lock acquisition to avoid the overhead of acquiring and releasing locks for every evicted item. The shadow queue is more challenging, since every evicted key needs to be installed in some application's shadow queue. Normally, any GET that results in a miss should check the application's shadow queue. So, blocking operations for the whole cleaning pass or even just for the whole duration needed to populate it with evicted keys would be prohibitive. Instead, the shadow queue is protected with a spin lock while it is being filled, but GET misses use a `tryLock` operation. If the `tryLock` fails, the shadow queue access is ignored.

The last point of synchronization is between GET operations and the cleaner. The cleaner never modifies the items that it moves. Therefore, GET operations do not acquire the lock to the segment list and continue to access records during the cleaning pass. This could result in a GET operation finding a reference in the hash table to a place in a segment that is cleaned before it is actually accessed. Memshare uses an epoch mechanism to make this safe. Each request/response cycle is tagged at its start with an epoch copied from a global epoch number. After a cleaning pass has removed all of the references from the hash table, it tags the segments with the global epoch number and then increments it. A segment is only reused when all requests in the system are from epochs later than the one it is tagged with.

5.4 Modularity

Memshare maintains separation between the cleaner and the arbiter. To do this, after a cleaning pass chooses segments, it notifies the arbiter which items are being cleaned. The arbiter ranks them and then calls back to the cleaner for each item that it wants to keep. If the relocation is successful, the arbiter updates the item's location in the hash table. Once the empty segments have been filled with relocated items, the arbiter removes the remaining entries from the hash table and updates per-application statistics and shadow queues. In this way, the cleaner is oblivious to applications, ranking, eviction policy, and the hash table. Its only task is efficient and parallel item relocation.

6 Evaluation

To understand Memshare's benefits, we ran two sets of tests. First, we ran a week-long multi-tenant Memcachier trace with Memshare to measure hit rate gains and end-to-end client-observed speedup. Second, we also benchmarked the implementation with the YCSB [20] workload to understand the overheads introduced by Memshare's online profiling and log cleaning.

Our experiments run on 4-core 3.4 GHz Intel Xeon E3-1230 v5 (with 8 total hardware threads) and 32 GB of DDR4 DRAM at 2133 MHz. All experiments are compiled and run using the stock kernel, compiler, and libraries on Debian 8.4 AMD64.

6.1 Performance

We tested the performance of Memshare using all the major applications from the Memcachier trace with the pooled memory and idle tax policies. Figure 5 presents

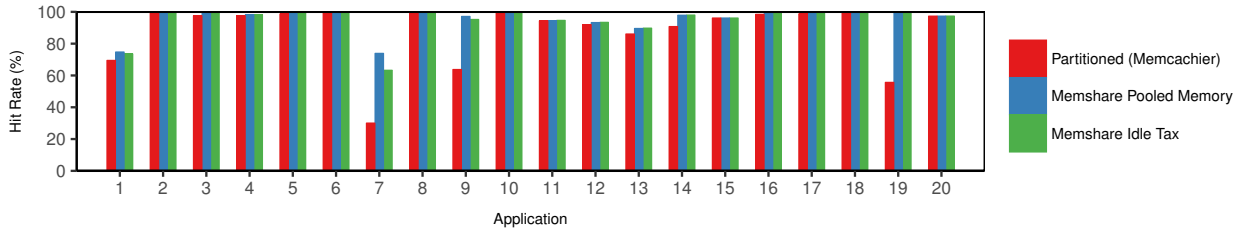


Figure 5: Memshare's pooled memory and idle tax algorithms' hit rates for top Memcachier applications compared to memcached.

Policy	Combined Hit Rate	Miss Reduction
memcached	84.66%	0.00%
Cliffhanger	87.73%	20.00%
Memshare Tax	89.92%	34.28%
Memshare Pooled	90.75%	39.69%

Table 6: Combined hit rate of Memshare's idle tax (50% tax) and pooled memory policy (75% reserved) compared with Cliffhanger, which is the state-of-the-art slab-based cache and Memcached. The miss reduction column compares the miss rate of the different policies to memcached.

the hit rate results, and Table 6 presents the summary. The pooled cache policy provides a higher overall combined hit rate increase, since it tries to maximize for overall hit rates. On average, Memshare reduces the number of misses by 39.7%. With an average cache latency of 100 µs and database latency of 10 ms, this would result in an average application-observed speedup of 1.59× (average access time of 1,016 µs versus 1,619 µs). In some cases, such as applications 7, 9, and 19, Memshare provides more than a 20% hit rate improvement.

Our evaluation uses 1 MB segments and 100 candidate segments for cleaning, the same as memcached's default slab and maximum item size. The number of candidate segments was chosen experimentally (see Table 7); it provides the best hit rate and results in less than 0.01% memory bandwidth use. The pooled policy used 75% of each application's original Memcachier memory as reserved with the rest pooled. Shadow queues were configured to represent 10 MB of items. Idle tax policy was set to a 50% tax rate with all memory reserved for each application. For the pooled policy, we experimented with different credit sizes. When credit sizes are too small, pooled memory isn't moved fast enough to maximize hit rates; when they are too high, memory allocation can oscillate, causing excessive evictions. We found a credit size of 64 KB provides a good balance.

Table 7 presents the combined hit rate and cleaner memory bandwidth consumption of Memshare's pooled memory policy when varying n, the number of segments that participate in each cleaning pass. The table shows that for the Memcachier traces, there is a diminishing increase in hit rate beyond n=40. While memory bandwidth use increases as the number of candidate segments is higher,

Segments (n)	Hit Rate	Memory Bandwidth (MB/s)
1	89.20%	0.04
10	90.47%	2.14
20	90.58%	2.86
40	90.74%	4.61
60	90.74%	6.17
80	90.75%	7.65
100	90.75%	9.17

Table 7: Combined hit rate and memory bandwidth use of top 20 applications in Memcachier trace using Memshare with the pooled memory policy with 75% reserved memory and varying the number of segments in each cleaning pass.

Policy	Average Single Tenant Hit Rate
memcached	88.3%
Cliffhanger	93.1%
Memshare 100% Reserved	95.5%

Table 8: Average hit rate of the top 20 applications in the trace run as a single tenant with Memshare with 100% reserved memory compared with Cliffhanger and memcached.

near peak hit rates can be achieved for this trace while consuming less than 0.01% of the memory bandwidth of a single modern CPU socket. Even at 100 candidate segments, the memory bandwidth of Memshare is less than 10 MB/s for the top 20 applications in the trace.

6.1.1 Single Tenant Hit Rate

In addition to providing multi-tenant guarantees, Memshare's log structured design significantly improves hit rates on average for individual applications on a cache which uses a slab allocator. Table 8 compares the average hit rates between Memshare and two systems that utilize slab allocators: memcached and Cliffhanger [19]. Within a single tenant application, Cliffhanger optimizes the amount of memory allocated to each slab to optimize for its overall hit rate. However, Memshare's log structured design provides superior hit rates compared to Cliffhanger, because it allows memory to be allocated fluidly for items of different sizes. In contrast, each time Cliffhanger moves memory from one slab class to another, it must evict an entire 1 MB of items, including items that may be hot. On average, Memshare with 100% reserved memory increases the hit rate by 7.13% compared to memcached and by 2.37% compared to Cliffhanger.

	Latency		
	GET Hit	GET Miss	SET
memcached	21.44 µs	21.8 µs	29.48 µs
Memshare	22.04 µs	23.0 µs	23.62 µs

Table 9: Memshare and memcached access latency under an artificial workload that causes high CPU load. Shadow queue lookups increases latency in the case of GET cache misses.

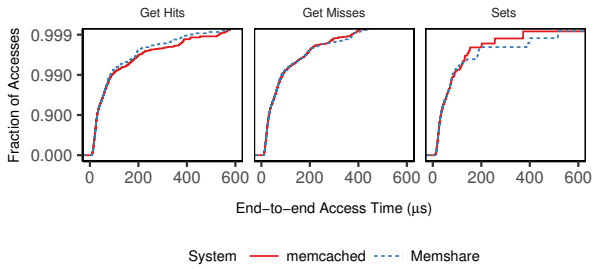


Figure 6: Tail latency distribution for Memshare/memcached.

6.2 Microbenchmarks

The Memcachier traces result in a low CPU utilization, so we also ran microbenchmarks using the YCSB framework [20] to stress CPU and memory bandwidth utilization. All of the microbenchmarks use 25 B items with 23 B keys over 100 million operations. Measurements always include the full cost of cleaning.

6.2.1 Latency

Table 9 shows the average response latency of Memshare with a full cache and a running cleaner compared to memcached. The clients and cache server are running on one machine, so the measurements represent a worst case. Access times are dominated by the network software stack and round trip delay [42]. Memshare's GET hit latency is 2.8% slower than memcached, and GET misses are 5.5% slower due to the check for the key in the shadow queue. Shadow queues are naïve LRU queues, so this could be mitigated. The additional latency on a miss is hidden, since the application must access the database which takes tens to hundreds of milliseconds.

Large-scale applications that exploit caches have high request fan-out and are known to be sensitive to tail latency [21, 39]. Figure 6 compares the tail latency of Memshare with memcached. Despite Memshare's slower average latency, it improves 99th and 99.9th percentile get hit response times from 91 to 84 µs and 533 to 406 µs, respectively. Get miss tail latency is nearly identical between the systems; despite the extra cost of maintaining the shadow queue, 99th and 99.9th percentile Memshare response times are 4 µs faster and 9 µs slower than memcached, respectively. 99th and 99.9th percentile set times show the impact of the cleaner with Memshare showing times 8 µs faster and 143 µs slower, respectively; most allocation is faster, but occasionally allocation is delayed by

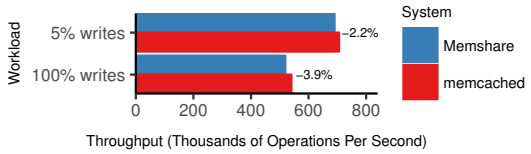


Figure 7: Average throughput of Memshare compared to memcached under a YCSB workload with 5% writes and 95% reads and under a worst case workload with 100% writes.

cleaning. Tail latency is often a concern for systems that perform garbage collection, like flash solid-state drives; Memshare is more robust against outliers since its critical sections are small and it never holds shared resources like serial channels to flash packages. Cleaning is fully parallel and effectively non-blocking.

6.2.2 CPU and Throughput

Table 7 compares Memshare throughput with memcached under a YCSB workload with 95%/5% reads/writes and one with 100% writes. Memshare is 2.2% slower for the first workload and 3.9% slower with all writes.

Most of the throughput loss is due to Memshare's cleaner. To breakdown the loss, we measured the CPU time spent on different tasks. In the 5% write workload, 5.1% of the process's CPU time is spent on cleaning, and 1.1% is spent testing shadow queues on GET misses. Note that the 100% write workload is unrealistic (such a workload does not need a cache). With a 100% write workload 12.8% of the process's CPU time is spent on cleaning.

The small decrease in Memshare's throughput is well justified. In-memory caches are typically capacity-bound not throughput-bound, and operate under low loads [16, 18]. The Memcachier trace loads are two orders of magnitude less than the microbenchmark throughput. Cache contents are often compressed; the gains from Memshare's efficient allocation are orthogonal, and the benefits can be combined since cleaning little CPU.

6.2.3 Memory Overhead and Utilization

Memshare has a small memory overhead. By default, shadow queues represent 10 MB of items; the overhead of the queues depends on the size of the items. Assuming small items on average (128 B), one queue stores 81,920 keys. Queues only keep 8 B key hashes, so key length isn't a factor. The default overhead is 81,920 · 8 B = 640 KB per application. The other structures used by Memshare have a negligible memory overhead.

Memshare's cleaner wastes some space by keeping some segments pre-cleaned; however, this space only represents about 1% of the total cache in our implementation. Even with some idle memory, Memshare is still better than memcached's slab allocator, since it eliminates the internal fragmentation that slab allocators suffer from. For example, in the trace, memcached's fragmentation restricts memory utilization to 70%-90%.

7 Related Work

Memshare builds on work in memory allocation and caching. Cliffhanger [19] estimated local hit rate curve gradients to rebalance slabs of items of different sizes. Memshare estimates local gradients to divide memory among applications. Memshare's log-structured allocator achieves significantly higher hit rates than Cliffhanger and flexibly moves memory across applications.

ESX Server [53] introduced idle memory taxation and min-funding revocation [52] in the context of a virtual machine hypervisor. Ranking functions to determine cache priorities were introduced by Beckmann et al [11] in the context of CPU caches. Memshare is the first application of both of these ideas to DRAM caches.

RAMCloud [45] and MICA [36] apply techniques from log-structured file systems [15, 37, 44, 47, 48] to DRAM-based storage. Log-structured caches have appeared in other contexts, such as a CDN photo cache [51] and mobile device caches [6]. Unlike these systems, Memshare addresses multi-tenancy. Also, MICA relies on FIFO eviction which suffers from inferior hit rates. Memshare enables application developers to apply any eviction policy using their own ranking functions.

MemC3 [23] and work from Intel [35] improve memcached multicore throughput by removing concurrency bottlenecks. These systems significantly improve performance, but they do not improve hit rates. In the case of Facebook and Memcachier, memcached is memory capacity bound, not CPU or throughput bound [16, 18].

Some caches minimize write amplification (WA) on flash [22, 51]. As presented, Memshare would suffer high WA on flash: low-need segments must be cleaned first, resulting in near-random 1 MB overwrites, which are detrimental for flash. Newer non-volatile media [2] may work better for Memshare.

Resource Allocation and Sharing. FairRide [43] gives a general framework for cache memory allocation and fairness when applications share data. Data sharing among competing applications is not common in key-value web caches. For both Facebook and Memcachier, applications each have their own unique key space; they never access common keys. For applications that do not share data, FairRide implements a memory partitioning policy in a distributed setup. Memshare, unlike FairRide, can efficiently use non-reserved and allocated idle memory to optimize the hit rate of applications and provide them with a memory boost in case of a burst of requests.

Mimir [46] and Dynacache [18] approximate stack distance curves of web caches for provisioning and slab class provisioning, respectively. They do not provide a mechanism for allocating memory among different applications sharing the same cache.

Efforts on cloud resource allocation, such as Moirai [50], Pisces [49], DRF [25] and Choosy [26] focus on performance isolation in terms of requests per second (throughput), not hit rate which is key in determining speedup in data center memory caches [16]. Similarly, there have been several projects analyzing cache fairness and sharing in the context of multicore processors [27, 30, 31]. In the context of multicore, fairness is viewed as a function of total system performance. Unlike CPU caches, DRAM-based web caches are typically separate from the compute and storage layer, so the end-to-end performance impact is unknown to the cache.

Ginseng [8] and RaaS [7, 13] are frameworks for memory pricing and auctioning for outsourced clouds; they only focus on pricing memory for applications that have dedicated memory cache servers running on VMs. In contrast, Memshare enables applications to share the same memory cache server, without the need for VM isolation. This is the preferred deployment model for most web application providers (e.g., Facebook, Dropbox, Box).

Eviction Policies. Many eviction schemes can be used in conjunction with Memshare. For example, Greedy-Dual-Size-Frequency [17] and AdaptSize [14] take into account request sizes to replace LRU as a cache eviction algorithm for web proxy caches. Greedy-Dual-Wheel [34] takes into account how long it takes to process a request in the database to improve cache eviction. EVA [10, 12] computes the opportunity cost per byte for each item stored in a cache. ARC [38], LRU-K [40], 2Q [29], LIRS [28] and LRFU [32, 33], offer a combination of LRU and LFU.

8 Conclusion

This paper demonstrates there is a large opportunity to improve key-value hit rates in multi-tenant environments, by utilizing dynamic and fungible memory allocation across applications. Memshare serves as a foundation for promising future research of memory sharing in distributed cache environments. For example, the techniques introduced in this paper are implemented within a single server running multiple applications. Similar techniques can be applied across servers, to determine the appropriate dynamic resources allocated to each application. Finally, key-value caches are being extended to other storage mediums beyond memory, such as flash and non-volatile memory. Running multiple applications on a heterogeneous caching environment creates novel future research challenges.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1566175. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Amazon Elasticache. aws.amazon.com/elasticache/.

[2] Intel® Optane™ Technology. http://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[3] Memcached. code.google.com/p/memcached/.

[4] Memcachier. www.memcachier.com.

[5] Redis Labs. www.redislabs.com.

[6] A. Aghayev and P. Desnoyers. Log-structured cache: Trading hit-rate for storage performance (and winning) in mobile devices. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '13, pages 7:1–7:7, New York, NY, USA, 2013. ACM.

[7] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. The rise of raas: the resource-as-a-service cloud. *Communications of the ACM*, 57(7):76–84, 2014.

[8] O. Agmon Ben-Yehuda, E. Posener, M. Ben-Yehuda, A. Schuster, and A. Mu'alem. Ginseng: Market-driven memory allocation. *SIGPLAN Not.*, 49(7):41–52, Mar. 2014.

[9] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[10] N. Beckmann and D. Sanchez. Bridging theory and practice in cache replacement. 2015.

[11] N. Beckmann and D. Sanchez. Modeling cache performance beyond LRU. *HPCA-22*, 2016.

[12] N. Beckmann and D. Sanchez. Maximizing cache performance under uncertainty. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 109–120. IEEE, 2017.

[13] O. A. Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. The Resource-as-a-Service (RaaS) cloud. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12)*, Boston, MA. USENIX.

[14] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter. AdaptSize: Orchestrating the hot object memory cache in a content delivery network. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 483–498, Boston, MA, 2017. USENIX Association.

[15] T. Blackwell, J. Harris, and M. I. Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *USENIX*, pages 277–288, 1995.

[16] N. Bronson. Personal Communication, 2016.

[17] L. Cherkasova. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Hewlett-Packard Laboratories, 1998.

[18] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA, July 2015. USENIX Association.

[19] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, Santa Clara, CA, Mar. 2016. USENIX Association.

[20] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[21] J. Dean and L. A. Barroso. The Tail at Scale. *Communications of the ACM*, 56:74–80, 2013.

[22] A. Eisenman, A. Cidon, E. Pergament, O. Haimovich, R. Stutsman, M. Alizadeh, and S. Katti. Flashield: a key-value cache that minimizes writes to flash. *CoRR*, abs/1702.02588, 2017.

[23] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 371–384, Berkeley, CA, USA, 2013. USENIX Association.

[24] B. Fitzpatrick. Distributed caching with Memcached. *Linux journal*, 2004(124):5, 2004.

[25] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 323–336, Berkeley, CA, USA, 2011. USENIX Association.

[26] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 365–378, New York, NY, USA, 2013. ACM.

[27] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '07, pages 25–36, New York, NY, USA, 2007. ACM.

[28] S. Jiang and X. Zhang. LIRS: An efficient low interreference recency set replacement policy to improve buffer cache performance. *SIGMETRICS Perform. Eval. Rev.*, 30(1):31–42, June 2002.

[29] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 439–450, 1994.

[30] H. Kasture and D. Sanchez. Ubik: Efficient cache sharing with strict QoS for latency-critical workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 729–742, New York, NY, USA, 2014. ACM.

[31] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 111–122, Washington, DC, USA, 2004. IEEE Computer Society.

[32] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *ACM SIGMETRICS Performance Evaluation Review*, volume 27, pages 134–143. ACM, 1999.

[33] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers*, (12):1352–1361, 2001.

[34] C. Li and A. L. Cox. GD-Wheel: a cost-aware replacement policy for key-value stores. In *Proceedings of the Tenth European Conference on Computer Systems*, page 5. ACM, 2015.

[35] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey. 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*, pages 476–488. ACM, 2015.

[36] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, Apr. 2014. USENIX Association.

[37] 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 *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 238–251, New York, NY, USA, 1997. ACM.

[38] N. Megiddo and D. S. Modha. Arc: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, pages 115–130, 2003.

[39] 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 V. Venkataramani. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[40] E. J. O'neil, P. E. O'neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD Record*, 22(2):297–306, 1993.

[41] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 29–41. ACM, 2011.

[42] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud Storage System. *ACM Transactions on Computer Systems*, 33(3):7:1–7:55, Aug. 2015.

[43] Q. Pu, H. Li, M. Zaharia, A. Ghodsi, and I. Stoica. FairRide: Near-optimal, fair cache sharing. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 393–406, Santa Clara, CA, Mar. 2016. USENIX Association.

[44] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[45] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured Memory for DRAM-based Storage. In *FAST*, pages 1–16, 2014.

[46] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson. Dynamic performance profiling of cloud caches. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 28:1–28:14, New York, NY, USA, 2014. ACM.

[47] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, pages 3–3. USENIX Association, 1993.

[48] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: A performance compari-

son. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, pages 21–21. USENIX Association, 1995.

[49] D. Shue, M. J. Freedman, and A. Shaikh. 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)*, pages 349–362, Hollywood, CA, 2012. USENIX.

[50] I. Stefanovici, E. Thereska, G. O'Shea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, and T. Talpey. Software-defined caching: Managing caches in multi-tenant data centers. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 174–181. ACM, 2015.

[51] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. RIPQ: Advanced photo caching on flash for Facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 373–386, Santa Clara, CA, Feb. 2015. USENIX Association.

[52] C. A. Waldspurger. Lottery and stride scheduling: Flexible proportional-share resource management. Technical report, Cambridge, MA, USA, 1995.

[53] C. A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, Dec. 2002.

Replication-driven Live Reconfiguration for Fast Distributed Transaction Processing

Xingda Wei, Sijie Shen, Rong Chen, Haibo Chen
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
Contacts: {rongchen, haibochen}@sjtu.edu.cn

ABSTRACT

Recent in-memory database systems leverage advanced hardware features like RDMA to provide transactional processing at millions of transactions per second. Distributed transaction processing systems can scale to even higher rates, especially for partitionable workloads. Unfortunately, these high rates are challenging to sustain during partition reconfiguration events. In this paper, we first show that state-of-the-art approaches would cause notable performance disruption under fast transaction processing. To this end, this paper presents DrTM+B, a live reconfiguration approach that seamlessly repartitions data while causing little performance disruption to running transactions. DrTM+B uses a pre-copy based mechanism, where excessive data transfer is avoided by leveraging properties commonly found in recent transactional systems. DrTM+B's reconfiguration plans reduce data movement by preferring existing data replicas, while data is asynchronously copied from multiple replicas in parallel. It further reuses the log forwarding mechanism in primary-backup replication to seamlessly track and forward dirty database tuples, avoiding iterative copying costs. To commit a reconfiguration plan in a transactionally safe way, DrTM+B designs a cooperative commit protocol to perform data and state synchronizations among replicas. Evaluation on a working system based on DrTM+R with 3-way replication using typical OLTP workloads like TPC-C and SmallBank shows that DrTM+B incurs only very small performance degradation during live reconfiguration. Both the reconfiguration time and the downtime are also minimal.

1 INTRODUCTION

Many applications like web services, stock exchange and e-commerce demand low-latency, high-throughput transactions over a large volume of data. Modern transaction processing systems scale by sharding data across a number of machines. State-of-the-art transaction processing systems like H-Store [16] and Silo [27, 30] have achieved orders of magnitude higher performance than previous systems. Recent designs like DrTM [29, 5], FaRM [11], and FaSST [15] further achieved millions of transactions per second on a small-scale cluster by exploiting novel hardware features like RDMA.

While sharding essentially distributes the client loads across multiple machines, it also faces a new challenge such that an improper data sharding scheme would cause notable workload imbalance as well as degraded overall performance. Work imbalance becomes even more severe for dynamically skewed workloads where the hotspot constantly changes over time [7, 3, 19, 17]. For example, the volume on the New York Stock Exchange (NYSE) during the first and last ten minutes of the trading day is an order of magnitude higher than at other times [24], while the access pattern is indeed skewed. This not only requires many distributed accesses in transactions but also causes frequent transaction aborts or stalls due to contended accesses on certain partitions. For example, our evaluation shows that when moving the warehouse selection from uniform to a highly skewed distribution, the transaction throughput degrades by 10X for TPC-C [26].

Prior approaches tackle this problem through *live reconfiguration* of the sharding plan [24]. Here, an optimal reconfiguration plan needs to balance among the following key requirements: 1) non-intrusiveness to running transactions; 2) minimal data movement; 3) balanced load after reconfiguration. To this end, E-Store [24] first generates an optimal plan according to current load distribution by re-assigning database tuples. It then uses Squall [12] to apply the new physical layout by migrating tuples online. Specifically, Squall fetches tuples on demand from the source node to reduce downtime. We term such an approach as a *post-copy* migration scheme, as an analogy to the live migration of virtual machines [6].

While post-copy approaches like Squall have been shown to be effective to quickly balance the load for H-Store [16], our investigation finds them ineffective for transaction processing systems with orders of magnitude higher throughput like DrTM [29, 5], FaRM [11] and FaSST [15]. This is due to prohibitive performance degradation during the lengthy post-copy phase, which is caused by enormous data transfer costs. Specifically, we have implemented the Squall-like approach on DrTM+R [5] and evaluated the effectiveness on TPC-C [26]. Even under configuration with low skew, there are near 4 seconds where millions of transactions execute with extremely low throughput and high latency, which means tens of millions of transactions were disrupted or stalled during the live reconfiguration process. This is prohibitively expensive for a transaction processing system demanding sub-millisecond latency and stable per-

formance. Consequently, this leaves the system trapped between two bad options: it can either delay reconfiguration and thus run with somewhat degraded performance for a long period, or it can reconfigure and thus run with seriously degraded performance for a short period.

This paper describes DrTM+B, a live reconfiguration scheme targeting distributed transaction processing with high throughput and low latency. Unlike prior post-copy based approaches [13, 12], DrTM+B uses a *pre-copy* based approach to reducing disruption to running transactions. Traditionally, pre-copy mechanisms [9, 10] would iteratively migrate tuples to the destination node (pre-copy phase) and only start the live reconfiguration process (commit phase) when the difference between the source and the destination nodes drops below a threshold, or the number of iterative copies exceeds a threshold. While such a pre-copy based approach causes little service disruption time, it may cause a notable downtime during the commit phase and the number of tuples to be transferred during the iterative pre-copy phase may still be non-trivial. To this end, DrTM+B also incorporates two key optimizations to further reduce data transfer and disruption to transactions.

First, we propose a novel reuse of fault-tolerant replicas to accelerate data transfer. This is based on the observation that state-of-the-art distributed transaction systems such as FaRM [11], FaSST [15] and DrTM+R [5] use primary-backup replication to tolerate failures. DrTM+B's reconfiguration plans take this into account: the new configuration uses previous backup nodes when possible to avoid physical tuple movement. This optimization can commonly avoid data copying in the pre-copy phase. When data copying is necessary, DrTM+B leverages all existing replicas by asynchronously pulling data in parallel. This can shorten the migration time and reduce disruption to transactions since the data copying uses one-sided RDMA operations to bypass the CPU of the source node, which may be busy with the transaction processing.

Second, data migration with pre-copy usually requires tracking and copying dirty data, even when the destination has already held a data copy (has a backup replica). This results in more migration iterations and lengthy downtime since many dirty tuples may be generated during the data migration process. To this end, DrTM+B reuses its fault-tolerance logging mechanism for tracking and copying dirty tuples. Therefore, no additional data copying is needed in the commit phase. Further, DrTM+B employs a cooperative commit protocol to minimize the downtime required to migrate final states, where the concurrency control protocol is slightly modified to be aware of the configuration change.

We have implemented DrTM+B by extending DrTM+R [5]. The extensions include reusing fault tolerance mechanism for live reconfiguration and making the OCC protocol aware of the reconfiguration process.[1] To demonstrate the effectiveness of DrTM+B, we evaluated it using TPC-C and SmallBank with changing skewness. Evaluation results show that DrTM+B can complete a reconfiguration within 40 milliseconds if existing replicas suffice to balance the workload. With data copying, DrTM+B only takes 3 seconds and 1 second to reconfigure TPC-C and SmallBank workloads with only 7% and 3% throughput drop during reconfiguration respectively. There is no observable downtime and DrTM+B finishes live reconfiguration significantly faster than existing post-copy approaches.

In summary, this paper makes the below contributions:

- A pre-copy based scheme to reduce downtime of live reconfiguration. (§3)
- Two key optimizations that further minimize data transfer, service disruption and downtime. (§4)
- An intensive evaluation that confirms the effectiveness of DrTM+B. (§6)

2 Background and Motivation

2.1 Fast In-memory Transaction Systems

Recent commoditization of advanced hardware features like RDMA have stimulated the design and implementation of fast distributed in-memory transaction systems like DrTM [29, 5], FaRM [11] and FaSST [15]. These systems exploit the low-latency transmission of RDMA to boost distributed transactions, resulting in orders of magnitude higher throughput compared to previous systems and reaching millions of transactions per second even on a small-scale cluster.

To achieve high throughput and low latency, such systems follow a local execution mode such that each worker thread will run a transaction to completion. The request will be routed to the node which contains most of the tuples the transaction accesses for efficiency. To scale, these systems continue the practice of H-Store [16] and others by splitting a large volume of data into multiple partitions spreading across multiple nodes.

To achieve high availability upon failures, a common approach is to use Vertical Paxos [18] with primary-backup replication [16, 11, 5, 15]. The primary-backup replication has been shown to have a lower number of replicas to tolerate the same number of failures compared to Paxos [11]. Under primary-backup replication, each partition is commonly configured to use 3-way replication (one primary and two backups). The transaction will write the log at each node with a backup before committing on the primary. To make backups catch up with

[1] DrTM+B can coexist with the fault-tolerance mechanisms based on primary-backup replication [5, 11] since both the pre-copy phase and the commit phase are applied in a transactionally safe way.

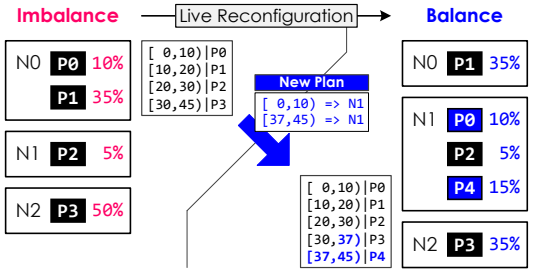


Fig. 1: *An example of reconfiguration.*

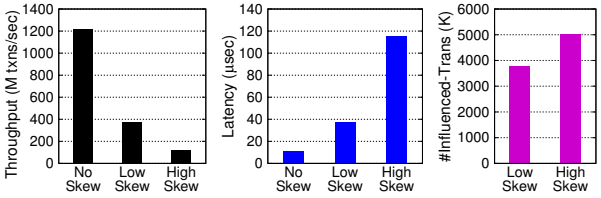


Fig. 2: *(a) The throughput and (b) the latency of DrTM+R, as well as (c) the number of influenced transactions during reconfiguration for TPC-C with different workloads.*

the primary, each node will periodically apply updates in logs to the backup replicas in background.

2.2 Skewed Workloads & Live Reconfiguration

While sharding scales out in-memory transactions, it is hard or even impossible to find a static sharding configuration that works well for all workloads, especially for those with dynamic changing hotspots and skewness [24]. Actually, prior work on production applications has shown that hot objects change rapidly over time [3, 7, 19, 17]. Some e-commerces like daily deals and flash sales can abruptly and significantly increase the visits and transactions on particular products, resulting in order spikes [1].

A change in skewness can cause severe load imbalance, leading to notably degraded overall performance. Fig. 1 shows a sample database which is partitioned into 4 partitions by the range of key and is initially assigned to 3 nodes with a balanced workload. However, the presence of dynamical changes in load may result in skewness, where some of the nodes will become overloaded while others may be idle. For example, if most accesses currently focus on the tuples in Partition 1 and 3 (P1 and P3), Node 0 and 2 (N0 and N2) will be overloaded while Node 1 (N1) will be underloaded.

To illustrate the performance impact from skewed workloads on fast distributed transaction systems, we conducted an initial experiment using DrTM+R [5] for TPC-C [26] on a 6-node cluster. For this experiment, TPC-C scales by partitioning a 25-million-tuples database into 192 warehouses (32 per node). We test different skewed settings (no skew, low skew and high skew), and report the average throughput and latency of the system. Additional details of our experimental setting are described in Sec. 6. As shown in Fig. 2(a), the throughput decreases by 3.3X and 10.0X from no skew to low skew and high skew respectively. In addition, as shown in Fig. 2(b), the increases of latency also reach 3.7X and 11.0X for low skew and high skew respectively.

Hence, it is highly desirable for a distributed in-memory transaction system to support fast and seamless live reconfiguration to quickly adapt to frequent workload changes. As shown in Fig. 1, the current hotspots on the sample database occur in Partition 1 and 3 (P1 with 35% and P3 with 50%). A proper reconfiguration plan is generated to lively migrate the Partition 0 and 4 (P0 and P4) from the overloaded nodes (N0 and N2) to the underloaded node (N1). To achieve this, some partition (i.e., P3) may need to be split to achieve fine-grained elasticity in the reconfiguration plan.

2.3 Disruptiveness of Post-copy Migration

To address the above issues, a recent state-of-the-art system called E-Store [24] has provided the live reconfiguration feature by automatically identifying whether a reconfiguration is needed and creating a new plan to balance the load across nodes. E-Store uses Squall [12] to execute the plan by lively migrating data among nodes in a transactionally safe way. Specifically, Squall follows the post-copy based approach [13], which first applies the reconfiguration plan and then pulls database tuples in an on-demand (reactive) way. It further introduces optimizations such as asynchronous migration and splitting reconfigurations.

Fig. 3 illustrates the timeline of the post-copy approach adopted by Squall. After receiving a new plan (migrating P0's primary from N0 to N1), the reconfiguration manager (RM) starts a reconfiguration by broadcasting the new plan to all nodes. All nodes then temporally suspend accesses to P0 and wait for all outstanding transactions on P0 to finish. After that, RM notifies all nodes to update the new location of P0's primary in the mapping table from partitions to nodes (PN table) and resume accesses to P0. Afterward, N0 will periodically migrate the tuples of P0 to N1 in an *asynchronous* way. Meanwhile, all transactions involving P0 will also be (re-)assigned to N1. N1 will examine whether the tuples the transaction accesses have been migrated. If not, N1 will block the transaction and send a pull request to *reactively* migrate the tuples from N0.

The post-copy scheme aims at reducing the downtime and avoiding repetitive data transfer. However, this causes significant disruption to transaction processing during data migration and a lengthy period of migration time due to two main reasons. First, transactions will be blocked due to missing database tuples. The transactions that access multiple tuples will mostly be blocked, even by multiple times. This is especially an issue for standard OLTP applications like TPC-C [26]. Second, migrat-

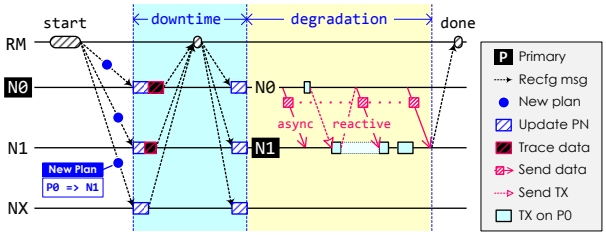


Fig. 3: *The execution flow of post-copy based reconfiguration.*

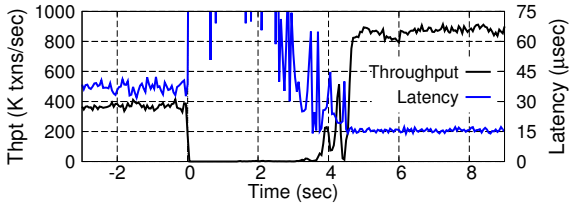


Fig. 4: *The throughput and median latency timeline of Squall live reconfiguration on DrTM+R for TPC-C with low skew.*

ing data will compete CPU and memory bandwidth with transaction processing on an already overloaded node, which may further aggravate the imbalance. The length and strength of performance degradation would be prohibitive for ultra-fast OLTP systems [5, 11, 15] that process millions of transactions per second with tens of microsecond latency.

To illustrate this, we implemented a post-copy based live reconfiguration on DrTM+R [5] with all optimizations in Squall [12], and conducted an experiment for TPC-C [26] with various workloads. Fig. 2(c) shows that the post-copy approach will disrupt several millions of transactions. Worse even, the throughput of DrTM+R degrades to nearly zero for more than 4 seconds when reconfiguring a TPC-C workload with *low skew*, as shown in Fig. 4. This is because the transactions in TPC-C require accessing multiple tuples, where fetching database tuples on demand would easily cause lengthy stalls and even aborts of transactions. Even worse, the length of performance degradation in the post-copy approach will proportionately increase with the growth of data size and the number of tuple accesses in a transaction. This becomes an even more serious issue for rapidly changing workloads that require frequent live reconfiguration [3, 7, 19, 17].

3 Overview of DrTM+B

DrTM+B is designed to support live reconfiguration for fast distributed in-memory transaction systems (§2.1). Like E-Store [24], DrTM+B contains two cooperative parts: Monitor and Planner. The monitor detects load imbalance and identifies the tuples causing it, and the planner decides whether there is a need to reorganize the tuples and generate the corresponding reconfiguration plan.

Unlike Squall [12], DrTM+B adopts a *pre-copy based* approach to migrating tuples, like those widely used in the live migration of virtual machines [6]. Fig. 5 illustrates a typical pre-copy mechanism which contains two consecutive phases: *iterative pre-copy* and *commit*. In the iterative pre-copy phase, the involved data is first copied from the source node to the destination node (i.e., DATA-COPY), then the dirty data is iteratively copied to the destination (i.e., DIFF-COPY). When the amount of dirty data is small enough or the number of iterations exceeds a threshold, the commit phase starts where it performs final data synchronization (i.e., DATA-SYNC) to transfer remaining dirty data, and state synchronization (i.e., STATE-SYNC) to (re-)assign transactions according to the new reconfiguration plan.

Issues with pre-copy. While the pre-copy based approach can minimize service disruption during data migration, there are two main issues limit its effectiveness in fast in-memory transaction systems. First, the pre-copy phase may be lengthy with large data transfer and may even hard to converge, since the high frequency of transaction processing will produce a huge amount of dirty tuples. Second, it is hard to find an efficient way to track the dirty tuples during migration and synchronize such tuples to destination nodes.

Observation. To address the above issues, we exploit two observations in fast in-memory transaction systems with high availability [11, 5, 15], where the primary-backup replication is used to tolerate failures. First, as the backup replicas contain nearly the same content as the primaries, it is possible to reuse the *fault-tolerant replicas* to avoid most data transfer in the pre-copy phase. Second, these systems typically use *log forwarding* to synchronize data between primaries and backups. It opens an opportunity to freely track and synchronize the updates on tuples during data migration.

Our approach. In the pre-copy phase (§4.2), DrTM+B prefers to reuse the *fault-tolerant replicas* to skip the DATA-COPY by a replication-aware reconfiguration plan (§5). Note that DrTM+B may split a partition when its granularity is not small enough to balance the workload (e.g., P3 in Fig. 1). For extremely rare cases where all nodes holding the replicas of a hot partition are also overloaded, DrTM+B will create a new replica for the hot partition at a spare node. Furthermore, since the source node holding the hot partition is busy with transaction processing, DrTM+B uses one-sided RDMA READ on the destination node to *asynchronously* pull tuples from all replicas in parallel. For the DIFF-COPY, DrTM+B seamlessly reuses the *log forwarding mechanism* in replication systems to freely track and synchronize the new updates on migrated tuples (i.e., dirty), since the log essentially contains the changes of each

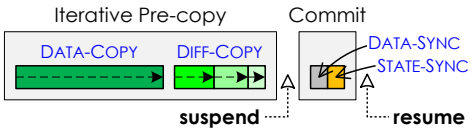


Fig. 5: *The execution flow of pre-copy live reconfiguration.*

committed transaction. Further, DrTM+B can concurrently execute DATA-COPY and DIFF-COPY.

In the commit phase (§4.3), DrTM+B supports the DATA-SYNC by *draining updates* in logs to the backup replica. When the amount of logs is too much and thus may cause a notable downtime, DrTM+B uses a *cooperative commit protocol* where a primary continues to commit transactions while the backup is applying logs. Yet, the transaction commit protocol (e.g., OCC) is modified such that the log versions of involved transactions are forwarded to the primary. Next, when the backup has applied all previous logs, it can quickly apply the remaining logs only synchronizing with the primary. In DrTM+B, the STATE-SYNC usually takes little time since it only needs to update a few state tables (e.g., PN table). Finally, one-sided RDMA READ is used to actively watch the state of the node holding the new primary, which helps timely resume the execution on migrated data.

4 REPLICATION-DRIVEN LIVE RECONFIGURATION

Our goals are to minimize service disruption and downtime while completing a reconfiguration as fast as possible. This section presents the detailed design on how DrTM+B optimizes both the pre-copy and the commit phases by a novel reuse of primary-backup replication.

4.1 Basic Data Structure

Database tuples in DrTM+B are assigned to multiple disjoint partitions, which are further distributed across multiple nodes. For brevity, this paper uses range partitioning as an example.[2] DrTM+B maintains a few data structures to support fine-grained live reconfiguration, as shown in Fig. 6. The first one is a mapping table from key to partition (KP table), which makes it possible to provide fine-grained reconfiguration at the tuple level. The second one is a mapping table from partition to node (PN table), which maintains the type (i.e., primary (P) or backup (B)) and the state (i.e., whether the transaction is executable (E) or not (N)) of each replica. Both KP and PN tables are replicated and individually changed on each node. A reconfiguration will change the two tables to reflect the new plan.[3]

[2]A two-tiered partitioning [24] can be used to fully support fine-grained planning and live reconfiguration at the tuple level.

[3]Each node updates the local tables atomically via an RCU-like mechanism during live reconfiguration and after all worker threads have executed at least one transaction.

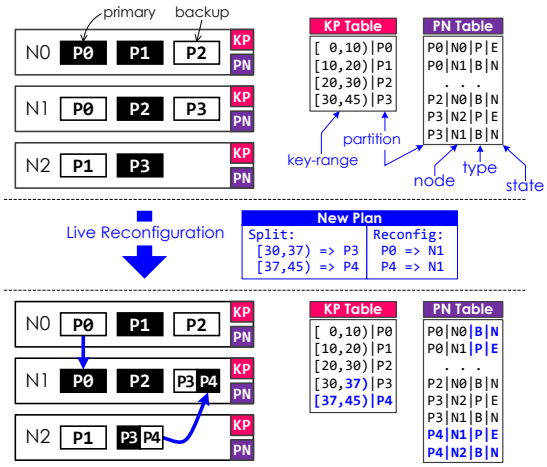


Fig. 6: *An overview of live reconfiguration on DrTM+B. In PN table, **P** and **B** stand for "Primary" and "Backup", which mean the type of the partition. **E** and **N** stand for "Executable" and "Non-executable", which mean the state of the partition. **C** and **X** stand for "Create" and "eXchange", which are used for creating partition and notifying transactions to enter exchange mode (see Fig. 7,8, 9, and 10).*

4.2 Pre-copy Phase

Generally, DrTM+B prefers to reuse existing replicas for fault tolerance to support live reconfiguration such that data transfer can be avoided. The underloaded nodes holding backup replicas are superior candidates to take over the workload from the overloaded nodes. However, for some highly skewed workloads, performing data migration at a partition granularity may not be able to achieve an optimal balance. Further, a spare node with the backup for the partition cannot always be found. DrTM+B addresses these two issues through *partition splitting* and *asynchronous replication* accordingly.

Partition splitting. To support fine-grained migration at *tuple level*, DrTM+B allows to split a single partition into multiple ones and can reconfigure these new partitions individually. For example, in Fig. 6, P3 is split into two new partitions (i.e., P3 and P4), and one of them (P4) is migrated from N2 to N1 in the commit phase.

Splitting a partition has minimal impact on outstanding transactions since there is no real data movement involved at this stage. Yet, we cannot naively split one partition. Consider the example in Fig. 6, where keys from [37,45) are being re-assigned from P3 to P4. If some transaction updates the key 40 in P3 on N2, while the key has been assigned to P4 at backup replica on N1, this replica may miss this update which causes inconsistency. It is vital to split the partition among all nodes *synchronously*. Moreover, all previous logs of committed transactions should be applied on all backups in advance. Consequently, DrTM+B defers the commit of splitting partitions to the commit phase (§4.3), which can synchronously change the configuration to the entire cluster.

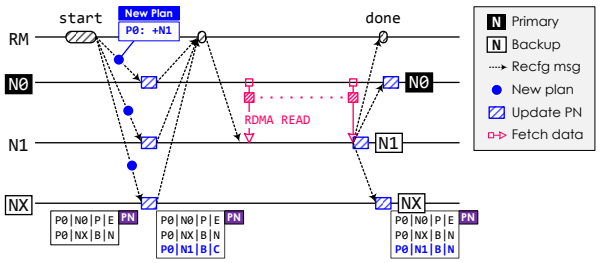


Fig. 7: *The execution flow of asynchronous replication. A new backup replica of P0 is created on N1.*

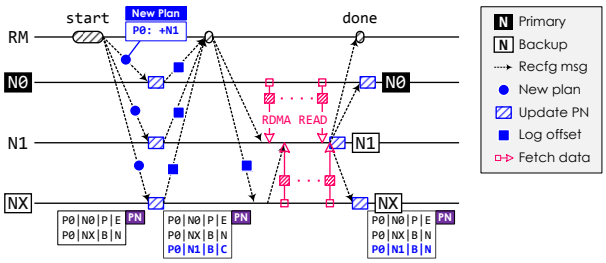


Fig. 8: *The optimized execution flow of asynchronous replication. A new backup replica of P0 is created on N1.*

Asynchronous replication. The number of replicas for fault tolerance depends on the requirement of degrees for availability. For 3-way replication, there are two candidate nodes that can be considered when reconfiguration without data movement. However, in some rare cases, the planner cannot find an optimal reconfiguration plan under current distribution of replicas. DrTM+B also allows to asynchronously create a new backup replica for some tuples in hot partitions on a spare node and then migrate the workload to that node.

To avoid adding new workload to hot nodes, DrTM+B adopts a pull-based approach, which leverages one-sided RDMA READ on the spare node to fetch tuples of the partition. To ensure the new backup replica receives the updates from running transactions, the reconfiguration manager (RM) will first acknowledge every node to add a new backup replica entry to its PN table. Consequently, subsequent transactions will write logs to the node which will hold the newly created replica. The spare node will start to fetch tuples from the primary of the partition after the notification from RM, which avoids missing some updates on the primary. Note that the spare node may fetch the tuples updated by transactions when creating the new replica. Yet, DrTM+B can still guarantee the consistency of the new backup even though the updates are duplicated in logs. Since each update has a version, the updates with an out-of-date version in the log or from the primary will be simply skipped. Finally, all nodes will receive the notification of the completion of asynchronous replication, and the new replica can be treated as a normal backup of the partition.

An example of asynchronous replication is shown in Fig. 7 where another copy of P0 is created on N1. For simplicity, we just create the whole P0 on N1 without losing generality. The process starts when RM sends the new plan to all nodes in the cluster. A new entry for N1's P0 with the state *Create* (**C**) is added to PN table when each node receives the new plan. After that, subsequent transactions will write logs to N1 after changing some tuples in P0, while N1 will drain updates in the log to the newly created backup. Each node replies to RM when all transactions have noticed the new plan. After all acknowledgments are collected by RM, RM then notifies N1 to fetch tuples from the primary of P0. At last, N1 will notify all nodes to update the state of N1's P0 to *Non-executable* (**N**) in their PN tables.

Parallel data fetching. While asynchronous replication has no interference with the execution of transactions, it may still delay the reconfiguration and increase the degree of imbalance. Since data fetching dominates the execution time of asynchronous replication, DrTM+B optimizes this by splitting the migrated data into multiple disjoint ranges and fetching them from multiple replicas in parallel. However, the backup replicas may be *stale* since some logs have not been drained. Hence, the RM must collect the latest log versions (i.e., log offsets) before changing PN table on all nodes[4] first, and then informs each backup to apply its logs beyond such versions. After that, the backups can allow the spare node to fetch data from them.

As shown in Fig. 8, the log offset of P0 on each node is piggybacked to the acknowledgment message to the RM, and the RM sends the collected log offsets to every node with the backup replica of P0 (NX in this example). After NX's logs have been drained according to the collected log offsets, NX sends an explicit message to N1 to invite it to fetch P0's data from its backup replica.

4.3 Commit Phase

The pre-copy phase guarantees that *a destination node has already possessed a backup replica with the migrated tuples, and new updates to these tuples are being forwarded through the primary-backup logs*. However, to commit a new configuration, it is necessary to atomically promote the backup to the new primary and demote the previous primary to the backup. Moreover, before exchanging the roles, DrTM+B must ensure that all updates in the log of the backup have been applied so that the primary and the backup are equivalent. Unfortunately, in common concurrency control protocols [11, 5], each transaction individually writes logs to all backup replicas of modified tuples. This means that each node contains the updates from all nodes and is not fully aware of which transaction wrote the updates and when.

[4]Each transaction will individually write the logs to all backups, such that each backup contains the logs from all nodes [11, 5, 15].

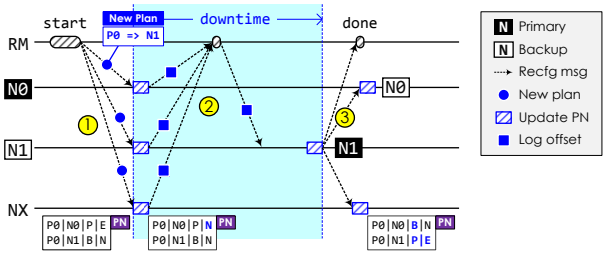


Fig. 9: *The execution flow of basic commit protocol.*

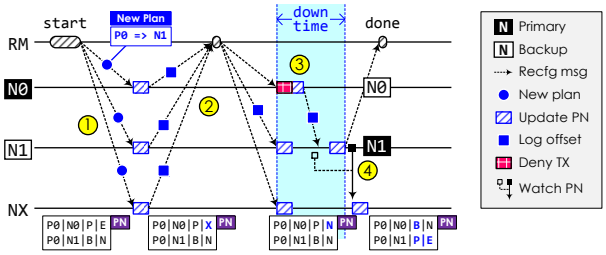


Fig. 10: *The execution flow of cooperative commit protocol.*

Basic Commit Protocol. The basic commit protocol will first suspend all involved transactions on every node, and then apply the logs to candidate backup replicas. Finally, all nodes will resume the involved transactions with new configuration. Fig. 9 shows the timeline of the basic commit protocol, where DrTM+B attempts to *exchange* the roles of P0's replicas on N0 (source) and N1 (destination) by executing the following steps:

1. *Suspend.* The RM will inform all nodes the new plan, and they first suspend all involved transactions on P0 by updating the state of P0's primary in PN table from *Executable* (**E**) to *Non-executable* (**N**).

2. *Collect.* Every node will send its log offset to the RM. The RM waits for all responses and informs the destination node to drain its logs according to the offsets.

3. *Resume.* After draining the logs, the destination node will inform the RM that the commit phase has done, and notify other nodes to resume the execution of involved transactions by updating their PN tables again. Each node will exchange the roles (i.e., primary and backup) of replicas and change the state of new primary from *Non-executable* (**N**) to *Executable* (**E**).

Note that the logs are continuously applied in background on each node. Hence, the logs on the destination node may have been applied beyond the log offsets before receiving the notification from the RM.

Cooperative Commit Protocol The downtime of the basic commit protocol highly depends on the number of updates in logs to be applied. Since the logs from the same node are stored together to be efficiently appended using one-sided RDMA WRITE, it is hard and not cost-effective to apply logs for a certain backup. Consequently, the destination node has to apply all logs from all nodes, which may result in a lengthy downtime of the commit phase when facing massive transactions.

Further, in DrTM+B and other systems [11, 5] which leverage RDMA primitives for log forwarding, the log versions are scattered over all nodes. Thus the log offsets must be collected from all nodes in the cluster to indicate the latest state of migrated tuples in the backup. Hence, the exchange of roles must be carried with a global coordination. Moreover, the destination node must notify all nodes to resume involved transactions after the commit phase, which may incur non-trivial downtime.

To remedy this, DrTM+B optimizes the basic exchange protocol to minimize the downtime by adding a new *exchange mode* for traditional concurrency control protocol (e.g., OCC). DrTM+B will ask all nodes running involved transactions in exchange mode by changing the state of primary in PN table from *Executable* (**E**) to *eXchange* (**X**). Under *exchange mode*, each node can still execute involved transactions but need to forward their log versions to the primary in addition. Then the primary can initiate the exchange and delivery the log offsets to the destination. The destination node will apply the logs and update the configuration in the PN table. The rest of nodes will watch the state of the new primary on demand and resume the execution of involved transactions. Fig. 10 shows the execution flow of the cooperative commit protocol, where DrTM+B attempts to exchange the roles of P0's replicas on N0 (source) and N1 (destination) by executing the following steps:

1. *Prepare.* The RM will inform all nodes the new plan, then every node notifies involved transactions to enter into the exchange mode by changing the state of P0's primary in its PN table from *Executable* (**E**) to *eXchange* (**X**). When the transaction executes in the exchange mode, it will explicitly inform the primary (N0) its log version during commitment with message. N0 will abort such transactions if it has already transferred its ownership.

2. *Collect.* Every node will send the log offset to the RM. The RM waits for all responses and informs the destination node to drain its logs according to the offsets.

3. *Suspend.* After receiving the message from RM, every node informs involved transactions to leave the exchange mode by setting the state of P0's primary to *Non-executable* (**N**). The primary (N0) starts to transfer P0's ownership to N1; it first denies transaction committing from other nodes and then informs N1 with the further log offsets collected from transactions running in the exchange mode.

4. *Resume.* To resume the execution on migrated data, N1 waits for the logs to be drained according to the offsets received from N0. Then it updates its PN table to resume involved transactions by exchanging the roles (i.e., primary and backup) of replicas and changing the state of new primary to *Executable* (**E**).

Unlike the basic commit protocol, DrTM+B leverages

an RDMA-friendly watching mechanism to timely notify all nodes to resume the execution on migrated data. The PN table will be allocated in an RDMA memory region which can be read by all nodes. Each node will lazily update its PN table until it watches that the partition has become *Executable* (**E**) at the destination node. For example, when a transaction on NX touches P0 whose state in PN table is *Non-executable* (**N**), NX will suspend the transaction and continuously probe N1's PN table until the state of P0 become *Executable* (**E**). Then NX will update its PN table to avoid further watching and resume the transaction.

ALGORITHM 1: Generate a reconfiguration plan.

```
   Data: L_N: an array of the workload for each node.
         L_P: an array of the workload for each partition.
         avg_load_N: the average load per node.
         P: a list of all partitions.
         T_PN: a mapping table from partitions to host nodes.
         (L_N, L_P and avg_load_N are provided by the monitor.)
 1: Function GENERATE_PLAN
 2:   new_plan ← {}
 3:   Sort P by the descending order in L_P
 4:   for p in P do
 5:     src = primary in T_PN[p]
 6:     if L_N[src] > avg_load_N then
 7:       dst = src, load = L_N[src]
 8:       for backup in T_PN[p] do
 9:         if (L_N[backup] < avg_load_N) &&
10:            (L_N[backup] < load) then
11:           dst = backup, load = L_N[dst]
12:       if dst != src then
13:         new_plan ← (p, dst)
14:         L_N[src] -= L_P[p], L_N[dst] += L_P[p]
15:   if HAS_BALANCED (L_N) then
16:     return new_plan
17:   ...// fine-grained planning of E-Store
```

5 Replication-aware Reconfiguration Plan

DrTM+B follows E-Store [24] to use a *two-phase monitoring* mechanism to detect load imbalance and identify the tuples causing it, as well as a greedy planning algorithm to generate the reconfiguration plan. To support the replication-driven live reconfiguration, DrTM+B extends the algorithm by considering the distribution of replicas.

The extended algorithm will first try to greedily balance the workload by migrating the partitions to the nodes that hold their replicas. As shown in Algorithms 1, the planner will start from the partition p with the highest workload $L_P[p]$ tracked by the monitor at runtime (line 3–4). If the node with the primary of partition p is overloaded (line 5–6), the planner will check whether the nodes with the backup of partition p are underloaded (line 7–11). If found, the node with the least workload will be designated as the destination of partition p in the reconfiguration plan (line 13). The workload of source and destination node will be updated (line 14). After iterating all partitions, the planner will return the new plan if the workload has been balanced (line 15–16). Otherwise, DrTM+B will follow the *fine-grained* planning algorithm in E-Store to refine the plan by splitting the partitions and/or creating new replicas, which requires the knowledge from *fine-grained* monitoring [24].

6 Evaluation

6.1 Experimental Setup

The performance evaluation was conducted on a small-scale cluster with 6 machines. Each machine has two 10-core Intel Xeon E5-2650 v3 processors with 128GB of DRAM and a ConnectX-3 MCX353A 56Gbps InfiniBand NIC via PCIe 3.0 x8 connected to a Mellanox IS5025 40Gbps InfiniBand Switch. All machines run Ubuntu 14.04 with Mellanox OFED v3.0-2.0.1 stack.

We implemented DrTM+B based on DrTM+R [5] where 3-way replication is enabled. Each machine dedicates one processor to run up to 8 worker threads and 2 auxiliary cleaner threads[5], the another processor is assigned to clients. To make an apple-to-apple comparisons, the state-of-the-art post-copy approach with all optimizations in Squall [12] was also implemented on DrTM+R as the baseline. In our experiments, we run all systems with 10s for warm-up and use a monitoring time window of 1s [24]. The backup replicas are randomly assigned to all nodes during database initialization.

We use two standard OLTP benchmarks to evaluate DrTM+B: TPC-C[26] and SmallBank [25]. *TPC-C* simulates principal transactions of an order-entry environment, which scales by partitioning a database into multiple warehouses spreading across multiple machines. We use a database with 192 warehouses (32 per node). The cross-warehouse ratio is set to 1% for the new-order transactions according to the specification. Similar to prior work [24], two skewed settings are evaluated. For low skew, the Zipfian distribution is used where two-thirds of transaction requests go to one-third of warehouses. For high skew, 40% of requests follow the Zipfian distribution used in low skew, while the remaining requests target four warehouses located initially on the first server.

SmallBank models a simple banking application where each transaction performs simple reads and writes on user accounts. SmallBank scales by partitioning user accounts into multiple partitions spreading across multiple machines. We use a database with 192 partitions (100K accounts for each). The default probability of cross-machine accesses is set to 1%. Again, two differ-

[5] DrTM+B follows DrTM+R to use auxiliary cleaner threads for applying updates in the log to the backup replicas periodically.

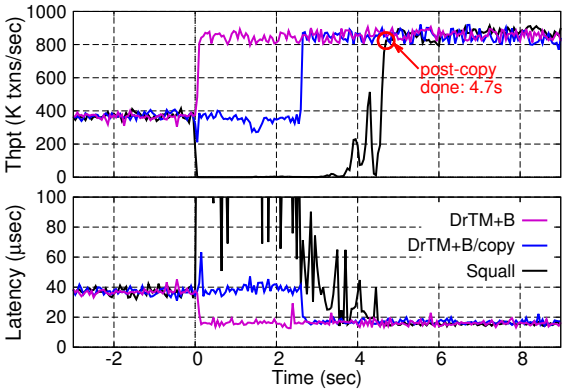


Fig. 11: *The perf. timeline for TPC-C with low skew.*

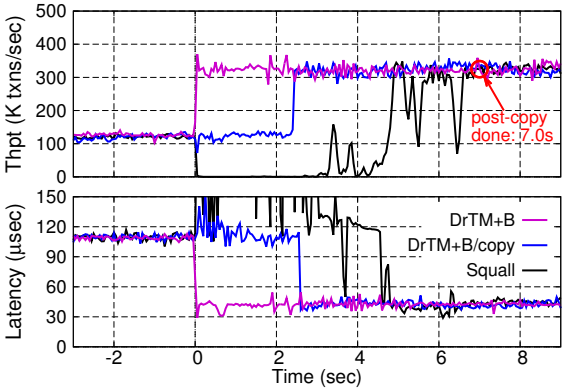


Fig. 12: *The perf. timeline for TPC-C with high skew.*

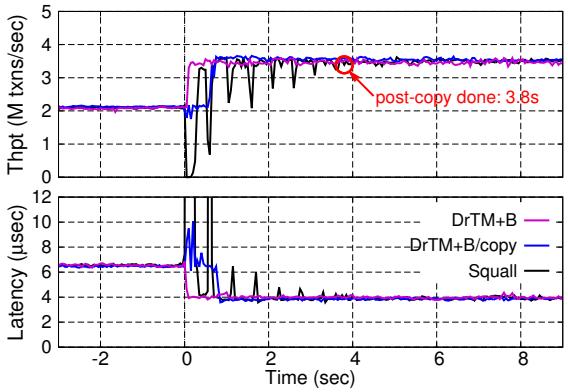


Fig. 13: *The perf. timeline for SmallBank with low skew.*

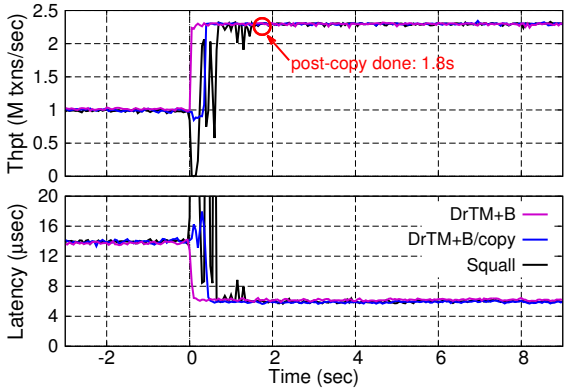


Fig. 14: *The perf. timeline for SmallBank with high skew.*

ent skewed settings are evaluated. For low skew, two-thirds of transaction requests go to 25% of records in the database. For high skew, 90% transaction requests go to 10% of records.

6.2 Performance for Skewed Workloads

We first evaluate the performance of DrTM+B and Squall for both benchmarks with various skewed workloads. The same initial configuration and reconfiguration plan are enforced to both systems, which ensures the same behaviors before and after live reconfiguration. In fact, for a 6-node cluster with 3-way replication, the existing backup is enough to provide load balance for both low and high skewed workloads. Therefore, DrTM+B can skip the data transfer. However, we still provide the performance of DrTM+B/copy as the reference, which enforces to migrate data with asynchronous replication to the destination node without regarding the existing data, while *parallel data fetching* and *cooperative commit protocol* are enabled by default.

TPC-C: Low skew. Fig. 11 shows the performance timeline of live reconfiguration with different approaches. After reconfiguration, the throughput increases by 2.1X, while the latency decreases by 33%. Since there is no data movement, DrTM+B has imperceptible downtime due to sub-millisecond commit phase. Squall needs 4.7s to finish reconfiguration due to *reactive* data transfer, while DrTM+B/copy only takes about 2.5s thanks to *parallel data fetching*. The performance degradation in DrTM+B/copy is also trivial, the average throughput drops by about 7% and the average latency increases by about 6% during live reconfiguration. This is because in DrTM+B/copy all logs are sent concurrently and data fetching mostly occupies CPU resources at *spare* nodes. For Squall, the throughput drops by around 99% and the latency increases by 7.7X in this period (the first 2.5s) due to frequent transaction aborts and increased contention on CPUs.

High skew. Fig. 12 shows how DrTM+B can improve the performance of TPC-C workload with high skew. After reconfiguration, the throughput increases by 3.0X, while the latency decreases by 64%. When there is no data movement, DrTM+B improves performance instantly. It takes about 7.0s for Squall, while DrTM+B/copy only needs 2.6s. Moreover the throughput of DrTM+B/copy decreases by about 2% during live reconfiguration and the latency increases by nearly 3%, while Squall suffers from 98% throughput degradation in this period since most workload is focused on hot spots.

SmallBank: Low skew. Fig. 13 shows the performance timeline of live reconfiguration with different approaches. After reconfiguration, the throughput in-

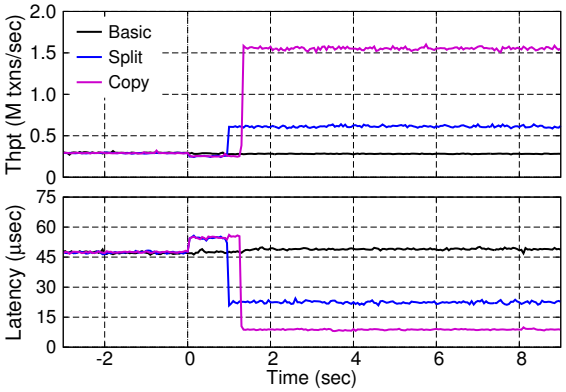


Fig. 15: *The perf. timeline for SmallBank with a load spike.*

creases by 1.7X, while the latency decreases by 42%. As existing backups are enough to balance, DrTM+B gains performance increase instantly. The reconfiguration by Squall requires 3.8s, while DrTM+B/copy only takes 0.7s to finish. Further, the throughput of DrTM+B/copy drops by only 3% and latency increases by 11% during live reconfiguration. By contrast, Squall has about 23% throughput drop and nearly 6.3X latency increase in this period. The throughput drops less for Squall in SmallBank compared with TPC-C since the transactions are much simpler and less affected by missing data.

High skew. Fig. 14 shows the performance timeline of SmallBank workload with high skew. After reconfiguration, the throughput increases by 2.4X and the latency decreases by 69%. Squall takes 1.8s to finishes while DrTM+B/copy only needs 0.3s. Moreover DrTM+B/copy has only 6% throughput drop with 8% latency increase without notable downtime, while the numbers for Squall are 22% and 3.8X respectively in this period. Although the performance is improving gradually during live reconfiguration, it still has non-negligible costs.

Load spike: We further evaluate DrTM+B with a load spike for SmallBank where 90% of workloads focus only on one partition (0.5% of records). In such workload data movement is necessary for the optimal plan. Fig. 15 shows the performance timeline of DrTM+B with different settings. Note that DrTM+B starts one second fine-grained monitoring after detecting the imbalance at time 0 to generate fine-grained plans for DrTM+B/split and DrTM+B/copy. This will tentatively increase latency by 17%. DrTM+B only exchanges primary with backup, which barely changes the performance since the hot data resides in one partition. DrTM+B/split uses *partition splitting* to balance the workload on the hot partition, which can immediately improve the throughput by 2.1X and reduce the latency by 54%. DrTM+B/copy further uses *asynchronous replication* to create new replicas on spare nodes, so that the throughput is improved by 5.4X

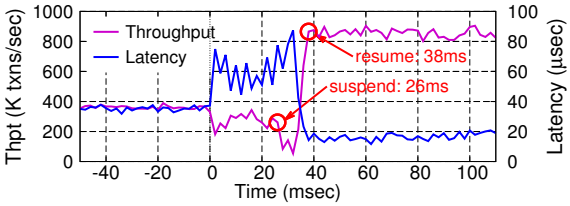


Fig. 16: *The focus timeline for TPC-C with low skew.*

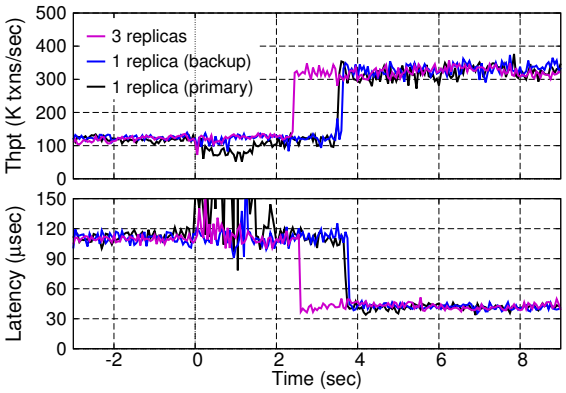


Fig. 17: *The perf. timeline using different replication copy mechanisms for TPC-C with high skew.*

while the latency is decreased by 82%. The improvement is slighted delayed (0.5s) due to data movement.

6.3 Breakdown of Commit Phase

Since the commit phase of the pre-copy approach will cause service downtime, we further study the time breakdown of DrTM+B's commit phase using TPC-C workload with low skew and illustrate the focus timeline in Fig. 16. RM issues a new plan at 0ms, suspends all nodes at 26ms, and resumes the execution at 38ms. In DrTM+B, all logs are concurrently drained during the commit phase, therefore the waiting time to activate the destination partitions is much short. Moreover, the downtime of DrTM+B is minimized to only 12ms thanks to the *cooperative commit protocol*. The throughput drops slightly by 12% during the commit phase.

6.4 Optimization for Replication Copy

To further study the impact of different optimizations on the pre-copy phase, we conduct an experiment on balancing TPC-C workload with high skew. Fig. 17 shows the performance timeline of live reconfiguration with three different settings in DrTM+B, which indicate fetching data directly from the primary, 1 backup replica, or all of 3 replicas in parallel. When fetching directly from the primary, it incurs 24% throughput drop and 2.8X latency increase due to the contention on CPU at the primary. Data fetching from 1 backup replica has nearly no impact to throughput, and the latency only increases 3%. This is because the backup node has less load than that of the primary. Moreover, data fetching from 3 replicas in parallel can notably shorten the migration time from

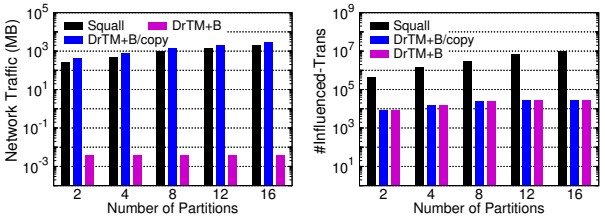


Fig. 18: *A comparison of (a) network traffic and (b) performance degradation between DrTM+B and Squall.*

3.5 to 2.6 seconds.

6.5 Micro-benchmark

We further use a micro-benchmark based on TPC-C to study the performance of DrTM+B and Squall, where 24 partitions are deployed on 3 nodes each holds one warehouse. We measure the metrics by swapping different numbers of partitions between first two nodes.

Network traffics. Fig. 18(a) shows the log scale data transferred with the increase of swapping partitions. For DrTM+B, only 4KB metadata is transferred during the commit phase since there is no data movement. For Squall, the network traffic is mainly dominated by the size of migrated partitions. For DrTM+B/copy, the data transferred is nearly doubled due to additional logs of the new backup replica, which can be avoided by removing one existing backup.

Influenced transactions. Fig. 18(b) presents the log scale influenced transactions with the increase of swapping partitions. DrTM+B and DrTM+B/copy have almost the same amount of influenced transactions, which only happens during the commit phase. The number of influenced transactions in DrTM+B is only 2% of Squall when swapping two partitions.

7 Related Work

Live reconfiguration for shared-storage database: There have been a few efforts to provide live reconfiguration features to shared-storage databases. For example, Albatross [9, 10] uses a traditional pre-copy approach to iteratively migrating the database. DrTM+B also uses a pre-copy approach but overcomes several limits through a novel reuse of fault-tolerant replicas. Zephyr [13] uses post-copy to migrate database while allowing transactions to execute during data migration. ProRea [20] extends Zephyr's approach by proactively migrating hot tuples to reduce service interruption. Elastras [8] decouples data storage nodes from transaction nodes and provides elasticity by moving or adding partitions. Slacker [4] leverages existing database backup tools to migrate data, and uses stop-and-copy or pre-copy method to create backups.

Live reconfiguration for partitioned databases: The importance of providing load balance has stimulated a few recent designs targeting partitioned databases. Squall [12] follows Zephyr's post-copy mechanism to performing live reconfiguration with the support of fine-grained tuple level migration. Compared to Zephyr, it introduces some optimizations such as pull prefetching and range splitting. In this paper, we show that using a post-copy based approach cause notable service disruption for fast in-memory transaction processing. DrTM+B makes a novel reuse and extension of the replication mechanism to achieve fast and seamless reconfiguration.

Providing elasticity on non-transactional systems: Spore [14] aims at addressing skewed access patterns on keys for in-memory caching systems (i.e., memcached). The basic approach is to create replicas of popular keys, which is similar to the asynchronous replication in DrTM+B. However, it does not consider the transactional execution on the key/value store. The reconfiguration in primary-backup systems has also been studied in distributed systems [28, 23]. For example, Shraer et al. [23] proposes a protocol to dynamically configure Apache Zookeeper without leveraging an extern reconfiguration service. Similarly, The commit protocol in DrTM+B incurs no downtime to partitions where their primaries have not changed during reconfiguration.

Generating reconfiguration plan: E-Store [24] proposes a fine-grained tracking approach and can generate a new partition plan to migrate data tuples between partitions. DrTM+B further extends it to consider the location of existing fault-tolerant replicas. Accordion [21] uses a mechanism to find a better partition plan to reduce distributed transactions. A few recent work has provided a general partitioning service for datacenter applications [2] or general DBMS schema [22].

8 Conclusion

This paper described DrTM+B, a fast and seamless live reconfiguration framework for fast in-memory transaction systems. It adopted a pre-copy based approach, which allows minimal interference between live reconfiguration and normal execution. DrTM+B further made a novel reuse of replication for fault tolerance in several ways to accelerate data transfer in pre-copy phase and minimize the downtime in commit phase. Evaluations using typical OLTP workloads with different skewed workloads confirmed the benefits of designs in DrTM+B.

Acknowledgments

We thank our shepherd Eddie Kohler 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. 61402284, 61572314, 61525204), the National Youth Top-notch Talent Support Program of China, and Singapore NRF (CREATE E2S2).

REFERENCES

[1] Daily Deals and Flash Sales: All the Stats You Need to Know. http://socialmarketingfella.com/daily-deals-flash-sales-stats-need-know/, 2016.

[2] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter, R. Peon, L. Kai, A. Shraer, A. Merchant, and K. Lev-Ari. Slicer: Auto-sharding for Datacenter Applications. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 739–753, Berkeley, CA, USA, 2016. USENIX Association.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS'12, pages 53–64, New York, NY, USA, 2012. ACM.

[4] S. Barker, Y. Chi, H. J. Moon, H. Hacigümüş, and P. Shenoy. "Cut Me Some Slack": Latency-aware Live Migration for Databases. In *Proceedings of the International Conference on Extending Database Technology*, EDBT'12, pages 432–443, New York, NY, USA, 2012. ACM.

[5] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and General Distributed Transactions using RDMA and HTM. In *Proceedings of the European Conference on Computer Systems*, EuroSys'16, page 26. ACM, 2016.

[6] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the USENIX Conference on Symposium on Networked Systems Design & Implementation*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[7] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: A Workload-driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow.*, 3(1-2):48–57, Sept. 2010.

[8] S. Das, D. Agrawal, and A. El Abbadi. ElasTraS: An Elastic, Scalable, and Self-managing Transactional Database for the Cloud. *ACM Trans. Database Syst.*, 38(1):5:1–5:45, Apr. 2013.

[9] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Live Database Migration for Elasticity in a Multitenant Database for Cloud Platforms. *CS, UCSB, Santa Barbara, CA, USA, Tech. Rep*, 9:2010, 2010.

[10] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration. *Proc. VLDB Endow.*, 4(8):494–505, May 2011.

[11] 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 *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP'15, pages 54–70, New York, NY, USA, 2015. ACM.

[12] A. J. Elmore, V. Arora, R. Taft, A. Pavlo, D. Agrawal, and A. El Abbadi. Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD'15, pages 299–313, New York, NY, USA, 2015. ACM.

[13] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD'11, pages 301–312, New York, NY, USA, 2011. ACM.

[14] Y.-J. Hong and M. Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proceedings of the Annual Symposium on Cloud Computing*, SOCC'13, pages 13:1–13:17, New York, NY, USA, 2013. ACM.

[15] A. Kalia, M. Kaminsky, and D. G. Andersen. 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*, OSDI'16, pages 185–201. USENIX Association, 2016.

[16] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: A High-performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.*, 1(2):1496–1499, Aug. 2008.

[17] A. Khandelwal, R. Agarwal, and I. Stoica. BlowFish: Dynamic Storage-Performance Tradeoff in Data Stores. In *13th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'16, pages 485–500, Santa Clara, CA, Mar. 2016. USENIX Association.

[18] L. Lamport, D. Malkhi, and L. Zhou. Vertical Paxos and Primary-backup Replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC'09, pages 312–313, New York, NY, USA, 2009. ACM.

[19] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware Automatic Database Partitioning in Shared-nothing, Parallel OLTP Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD'12, pages 61–72, New York, NY, USA, 2012. ACM.

[20] O. Schiller, N. Cipriani, and B. Mitschang. ProRea: Live Database Migration for Multi-tenant RDBMS with Snapshot Isolation. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT'13, pages 53–64, New York, NY, USA, 2013. ACM.

[21] M. Serafini, E. Mansour, A. Aboulnaga, K. Salem, T. Rafiq, and U. F. Minhas. Accordion: Elastic Scalability for Database Systems Supporting Distributed Transactions. *Proc. VLDB Endow.*, 7(12):1035–1046, Aug. 2014.

[22] M. Serafini, R. Taft, A. J. Elmore, A. Pavlo, A. Aboulnaga, and M. Stonebraker. Clay: Fine-grained Adaptive Partitioning for General Database Schemas. *Proc. VLDB Endow.*, 10(4):445–456, Nov. 2016.

[23] A. Shraer, B. Reed, D. Malkhi, and F. P. Junqueira. Dynamic Reconfiguration of Primary/Backup Clusters. In *Proceedings of the USENIX Annual Technical Conference*, USENIX ATC'12, pages 425–437, 2012.

[24] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-Store: Fine-grained Elastic Partitioning for Distributed Transaction Processing Systems. *Proc. VLDB Endow.*, 8(3):245–256, Nov. 2014.

[25] The H-Store Team. SmallBank Benchmark. http://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/.

[26] The Transaction Processing Council. TPC-C Benchmark V5.11. http://www.tpc.org/tpcc/.

[27] 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.

[28] R. Van Renesse and F. B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation*, volume 4 of *OSDI'04*, pages 91–104, 2004.

[29] 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.

[30] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 465–477. USENIX Association, 2014.

HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems

Fei Xia[1,2], Dejun Jiang[1], Jin Xiong[1], and Ninghui Sun[1]

[1]SKL Computer Architecture, ICT, CAS [2]University of Chinese Academy of Sciences
{*xiafei2011,jiangdejun,xiongjin,snh*}*@ict.ac.cn*

Abstract

Hybrid memory systems consisting of DRAM and Non-Volatile Memory are promising to persist data fast. The index design of existing key-value stores for hybrid memory fails to utilize its specific performance characteristics: fast writes in DRAM, slow writes in NVM, and similar reads in DRAM and NVM. This paper presents HiKV, a persistent key-value store with the central idea of constructing a hybrid index in hybrid memory. To support rich key-value operations efficiently, HiKV exploits the distinct merits of hash index and B$^+$-Tree index. HiKV builds and persists the hash index in NVM to retain its inherent ability of fast index searching. HiKV builds the B$^+$-Tree index in DRAM to support range scan and avoids long NVM writes for maintaining consistency of the two indexes. Furthermore, HiKV applies differential concurrency schemes to hybrid index and adopts ordered-write consistency to ensure crash consistency. For single-threaded performance, HiKV outperforms the state-of-the-art NVM-based key-value stores by reducing latency up to 86.6%, and for multi-threaded performance, HiKV increases the throughput by up to 6.4x under YCSB workloads.

1 Introduction

Emerging Non-Volatile Memory (NVM) technologies, such as PCM [1], ReRAM [2], and the recent 3D XPoint [3], are drawing substantial attentions from both academia and industry. One potential opportunity of NVM is to act as a fast persistent memory sitting on the memory bus, leading to hybrid DRAM-NVM memory systems [4, 5, 6]. Building storage systems, such as key-value stores, towards hybrid memory allows one to exploit fast memory access to achieve improved performance compared to basing on traditional hard disks or flash-based solid state drives (SSDs).

Persistent key-value stores (KV stores) have become an important part of storage infrastructure in data centers. They are widely deployed in large-scale production environments to serve search engine [7, 8], e-commerce platforms [9], social networking [10, 11], photo stores [12, 13], and more. In the past decade, there has been a large body of research on KV store design and optimization, on topics such as reducing write amplification to SSDs [14, 15, 16], reducing memory usage of indexing [17, 18, 19], and improving concurrency to achieve high scalability [11, 20, 21, 22]. Conventional KV stores are not suitable for hybrid memory systems because they are designed for the performance characteristics of hard disks or SSDs. For instance, many of existing studies adopt Log-Structured Merge Tree as the indexing structure [8, 11, 14, 15, 16], which avoids small random writes to hard disks or SSDs. Differing from hard disks and SSDs, hybrid memory systems are byte-addressable, and provide similar performance for sequential and random access. Maintaining sequential writes in large granularity instead introduces write amplification to NVM when designing KV stores for hybrid memory systems.

Indexing is a fundamental issue in designing key-value stores. The efficiency of supporting rich KV operations, such as *Put, Get, Update, Delete,* and *Scan*, is largely decided by the operational efficiency of indexing structure. For instance, searching B$^+$-Tree index is usually more costly than searching hash index. As we will show in Section 2.2, the operational efficiencies of different indexing structures are largely varied. Recently, a number of optimizations on B$^+$-Tree index are proposed for NVM memory systems [23, 24, 25, 26, 27, 28, 29, 30]. However, these techniques mainly focus on reducing consistency cost when directly persisting B$^+$-Tree index in NVM. On the other hand, the scalability of key-value stores is limited by the scalability of the indexing structure. For instance, partitioning the hash index allows one to scale the indexing structure to multiple threads, but partitioning the B$^+$-Tree index incurs expensive data movement when splitting large partitions or merging small ones. Thus, we argue that the choice of indexing structure for designing KV stores on hybrid memory is still open.

In this paper, we propose HiKV, a Hybrid index Key-Value store to run on hybrid memory. The central idea behind HiKV is the adoption of hybrid index: a hash index placed and persisted in NVM, and while a B$^+$-Tree index placed in volatile but fast DRAM without being persisted. The hybrid index fully exploits the distinct merits of the two indexes. It retains the inherent efficiency of hash operations to support single-key operations (*Get/Put/Update/Delete*). Moreover, it efficiently

accelerates *Scan* using the sorted indexing in B$^+$-Tree.

Adopting hybrid index introduces a number of challenges. First, when serving certain KV operations, including *Put*, *Update*, and *Delete*, the latency can be increased as HiKV needs to update two indexes to keep them consistent. HiKV solves this by placing the slow B$^+$-Tree index in fast DRAM and the fast hash index in slow NVM. In addition, HiKV updates the B$^+$-Tree index asynchronously to further hide its latency. Second, the scalability of the hybrid index requires careful design. Partitioning the hash index provides good scalability, while partitioning the B$^+$-Tree index suffers from high cost due to data migration. HiKV thus adopts partitioned hash indexes and a global B$^+$-Tree index. HiKV applies Hardware Transactional Memory (HTM) for the concurrency control of B$^+$-Tree index, and fine-grained locking to support concurrent accesses within individual hash index partitions. Finally, guaranteeing crash consistency of the hybrid index incurs expensive writes to NVM. HiKV adopts selective consistency that only ensures the consistency of hash index and key-value items by ordered-write. HiKV keeps the B$^+$-Tree index in DRAM and rebuilds it after system failure.

We implement HiKV and the state-of-the-art NVM-based key-value stores NVStore [28] and FPTree [30]. We evaluate the three KV stores using both micro-benchmarks and the widely used YCSB. For micro-benchmarks, HiKV can reduce latency by 54.5% to 83.2% and 28.3% to 86.6% compared with NVStore and FPTree, respectively. For YCSB workloads, HiKV outperforms NVStore by 1.7x to 5.3x, and FPTree by 24.2% to 6.4x in throughput.

This paper makes the following contributions:

1. We propose a hybrid index consisting of a hash index in NVM and a B$^+$-Tree index in DRAM to fully exploit the performance characteristics of hybrid memory to efficiently support rich KV operations.
2. We carefully design different concurrency schemes for the hybrid index to achieve high scalability with partitioned hash indexes and single global B$^+$-Tree index.
3. We propose ordered-write consistency and specific hash index design allowing atomic writes to ensure the crash consistency with reduced NVM writes.
4. We implement HiKV on top of the hybrid index. We conduct extensive evaluations to show the efficiency of the design choices of HiKV.

2 Background and Motivation

2.1 Non-Volatile Memory

Emerging Non-Volatile Memory (NVM) technologies, such as Phase Change Memory (PCM) [1] and Resistive Memory (ReRAM) [2], can provide faster persistence than traditional Disk and Flash. Table 1 shows the performance characteristics of different memory technologies. NVM provides similar read latency to DRAM, while its write latency is apparently longer than DRAM. Similar to NAND Flash, the write endurance of NVM is limited, especially for PCM. Thus, reducing writes to NVM is critical for software system design. At last, NVM has high performance for random accessing like DRAM, which is different from traditional Flash.

Table 1: Characteristics comparison of different memory technologies [28, 31, 32, 33, 34]

Category	Read latency	Write latency	Write Endurance	Random accessing
DRAM	60ns	60ns	10^{16}	High
PCM	50∼70ns	150∼1000ns	10^{9}	High
ReRAM	25ns	500ns	10^{12}	High
NAND Flash	35us	350us	10^{5}	Low

2.2 KV operations and indexing efficiency

The *Put*, *Get*, *Update*, and *Delete* are basic operations for KV stores. Besides, the *Scan* (short name for *Range Scan*) becomes important as required by today's applications. For instance, Facebook has replaced the storage engine of MySQL with a KV store MyRocks [35]. *Scan* turns out to be an important operation to serve range query of MySQL. Local file systems (i.e. TableFS [36]) and distributed file systems (i.e. CephFS [37]), use KV stores to store metadata. *Scan* is the core operation to support the second most prevalent metadata operation *readdir* [38]. Thus, efficiently supporting rich KV operations is significant for building key-value stores.

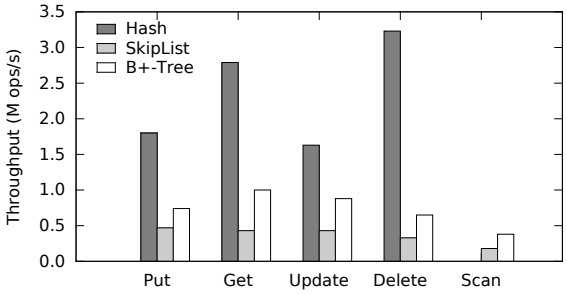


Figure 1: Throughput of different indexes

However, neither the hash indexing nor the sorted indexing can efficiently support all these operations. We use micro-benchmarks to quantify the efficiency of three widely used in-memory indexes: hash, skiplist, and B$^+$-Tree, to support the five KV operations. Figure 1 shows the in-memory throughput results with 50M key-values. For *Put/Get/Update/Delete*, hash index performs the most efficiently compared to the other two in-

dexes. Hash index usually involves less memory operations than skiplist and B$^+$-Tree, which requires multiple levels searching. However, as a non-sorted indexing, hash index provides extremely low throughput for *Scan* due to the cost of scanning the whole index space. Current NVM-based KV stores follow the widely adoption of B$^+$-Tree as the indexing structure. However, the above results motivate us to propose hybrid index to exploit distinguished merits of different indexes.

3 HiKV Design and Implementation

In this section, we present the system design and implementation of HiKV. We first present the design of the hybrid index. We then describe design issues when adopting hybrid index, including index updating, concurrency control, and crash consistency guaranteeing. Following that, we present the recovery of HiKV. At last, we describe the implementation of HiKV.

3.1 Hybrid index

Basic key-value operations include *Put, Get, Update, Delete,* and *Scan*[1]. To locate the requested key-value item, the single-key operations (*Put/Get/Update/Delete*) first take exactly one key to search the index. Once the KV item is located, *Get* directly returns the data, and while the write operations (*Put/Update/Delete*) require to persist updated index entry and new KV item if provided. Thus, the efficiency of index searching and data persisting is significant for these operations. Hash indexing inherently supports highly-efficient searching. Besides, regarding NVM reads perform similarly as in DRAM, placing a hash index in NVM as part of the hybrid index is a reasonable design choice. This design not only retains fast searching of hash index, but also allows persisting index in NVM directly without extra data copy from DRAM to NVM.

On the other hand, *Scan* takes a start key and count (or a start key and an end key) as input, which can benefit from sorted indexing. To efficiently support *Scan*, the hybrid index employs the widely used B$^+$-Tree index in main-memory systems [39, 40]. To maintain a consistent hybrid index, updating both hash index and B$^+$-Tree index for KV writes is fundamentally required. Updating B$^+$-Tree index involves many writes due to sorting as well as splitting/merging of leaf nodes. We thus place the B$^+$-Tree index in fast DRAM to avoid slow NVM writes in hybrid memory.

Figure 2 shows the architecture of hybrid index in hybrid memory. We discuss the issue of hybrid index updating in Section 3.2. Furthermore, to serve concurrent requests, the hash index is divided into multiple partitions. The B$^+$-Tree index is instead designed as a global one to indexes all KV items. We explain the differential concurrency schemes in Section 3.3. HiKV only guarantees the consistency of KV items and the hash index for improved performance. We present the ordered-write consistency mechanism in Section 3.4.

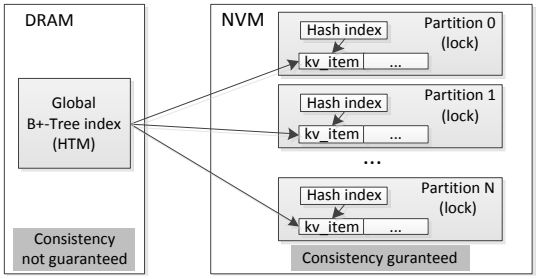


Figure 2: Architecture of hybrid index

3.2 Index updating

3.2.1 Asynchronous updates

When serving KV writes (*Put/Update/Delete*), HiKV needs to update both the hash index and the B$^+$-Tree index to keep them in a consistent state. One intuitive solution is to synchronously update both indexes. Due to the costly tree structure-specific operations, such as searching, sorting, splitting and merging, synchronous updates for the B$^+$-Tree index add extra latency to KV writes. Thus, HiKV employs asynchronous updates for hybrid index. In other words, HiKV retains synchronous updates to KV items and the hash index in NVM. For the B$^+$-Tree index in DRAM, HiKV asynchronously updates it in the background to hide the extra latency.

Figure 3 shows the procedure of HiKV to serve different KV operations. Taking *Put* as an example. HiKV first uses a *serving thread* to serve the incoming request. The serving thread is responsible for writing KV items to NVM (step1), and then writing the newly-added index entry to the hash index (step2). At last, the serving thread inserts the *Put* request to an updating queue (step3) and then returns. An asynchronous thread (called *backend thread*) gets requests from the updating queue and operates the B$^+$-Tree index in the background. In case of failing to update the B$^+$-Tree index due to system crash, HiKV can recover the B$^+$-Tree index from the hash index as presented in Section 3.5. In such doing, the observed latency of KV writes can be reduced.

However, a *Scan* request faces an inconsistent state of the B$^+$-Tree index as long as there exists requests in the updating queue when it arrives. Directly serving the scan request would retrieve old or invalid data. HiKV solves this by temporally blocking subsequent writes to enter into the updating queue once a scan is received. The

[1]Existing KV stores (i.e. Redis) support batch operations, such as MutiPut and MultiGet. HiKV can be extended to support such batch operations, which we leave as our future work.

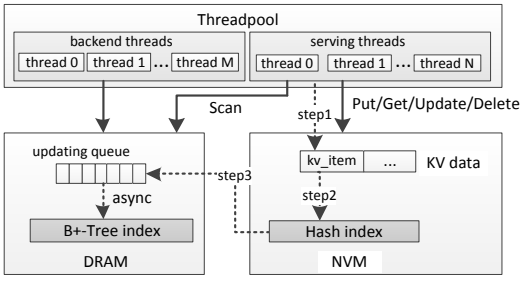


Figure 3: Procedure to serve KV operations

scan and subsequent writes wait until when all existing requests in the updating queue are processed. Once the updating queue becomes empty, it starts to receive further requests, and meanwhile the scan is served. Then, the concurrency control among the scan and subsequent writes on the B$^+$-Tree index are provided by Hardware Transactional Memory (HTM). We limit the length of the updating queue (i.e., holding 4096 requests in this paper) to avoid excessive waiting.

3.2.2 Dynamic threads adaption

To serve highly-concurrent requests, HiKV needs to increase the number of serving threads. For read-write mixed workloads, this can rapidly fill the updating queue as many serving threads put write requests into the queue. If the backend threads fall behind the serving threads, the updating queue becomes full and further blocks serving subsequent requests. Thus, HiKV needs to dynamically adapt the backend threads according to the change of serving threads.

We usually set a fix-sized thread pool to run both serving threads and backend threads. The dynamic threads adaption scheme is to decide the numbers of serving threads (N_{sthd}) and backend threads (N_{bthd}). Basically, we need to match the average processing rate of the backend threads on the updating queue with the average queue filling rate of the serving threads. The processing rate and filling rate are determined by a number of factors, such as KV operation complexity, ratio of different KV operations, and DRAM/NVM performance. To decide N_{sthd} and N_{bthd} at runtime, we sample the numbers of different KV operations as well as their average operational latencies. Based on our observation, the operational latency of *Scan* is 14 times than that of *Get*, and the latency gap among *Put*, *Update*, and *Delete* is less than 2x. For simplicity, we do not distinguish *Put/Update/Delete* when sampling but sample *Scan* and *Get* operations separately. Within each sampling window, assuming the number of writes is normalized to 1, the number of *Get* is N_g and the number of *Scan* is N_s. The average latencies of *Get* and *Scan* are L_g and L_s. The average write latencies of backend thread and serving thread are L_{bw} and L_{sw}, respectively. Then, N_{sthd} and N_{bthd} should satisfy the following two equations, in which N_{thd} is the total size of the thread pool. In such doing, the average processing latency of backend threads matches the one of serving threads.

$$(N_g \cdot L_g + N_s \cdot L_s + 1 \cdot L_{sw})/N_{sthd} = (1 \cdot L_{bw})/N_{bthd} \quad (1)$$

$$N_{sthd} + N_{bthd} = N_{thd} \quad (2)$$

3.3 Differential concurrency

Concurrency control is a key issue for improving the scalability of KV stores in the multi-core era. In this section, we present the differential concurrency schemes applied to the hybrid index.

Partitioning is shown to achieve high throughput and scalability for balanced workloads [40]. Thus, HiKV adopts the widely-used keyhash-based partitioning [41, 37, 42] for the hash index. All KV items are distributed to multiple partitions according to the hash value of the key, and each partition uses a hash index as Figure 2 shows. HiKV allows concurrent accessing to a partition by multiple threads to handle skewed workloads. It uses fine-grained locking for concurrency control inside each partition. HiKV uses an atomic write to update the hash index entry as illustrated in Section 3.4. As a result, HiKV can read an index entry when another thread is updating it without locking.

Partitioning the B$^+$-Tree index results in either *unordered multi-B$^+$-Tree indexes* as in Cassandra [43] and Megastore[44] using keyhash-based approach, or *ordered multi-B$^+$-Tree indexes* as in SLIK [45] using range partition. However, we argue that none can efficiently support *Scan* due to extra efforts. With unordered multi-B$^+$-Tree indexes, we need to issue the scan request to all indexes, and then return the matching key-values from the result. Such approach increases the concurrency overhead. With ordered multi-B$^+$-Tree indexes, the scan request can be only issued to indexes that contain corresponding KV items. However, such approach needs to migrate index entries when an index becomes too large or too small, which degrades system performance. Thus, HiKV adopts a global B$^+$-Tree index for all KV items in NVM. HiKV adopts HTM to handle concurrency control of the global B$^+$-Tree index.

3.4 Ordered-write consistency

Guaranteeing crash consistency is a fundamental requirement for persistent KV stores. Since NVM has long write latency, HiKV needs to reduce NVM writes when guaranteeing consistency. We first apply selective consistency to HiKV to only ensure the consistency of hash index and KV items, but not guarantee consistency for the B$^+$-Tree index to avoid expensive data copy from

DRAM to NVM. Upon a system failure, HiKV recovers the B$^+$-Tree index as presented in Section 3.5.

Secondly, we apply ordered-write to ensure the consistency of the hash index and KV items. Conventional logging and copy-on-write incur two writes when guaranteeing consistency. The ordered-write consistency first updates the KV item out-of-place. Then, it updates the hash index in-place using an atomic write. A KV item is not visible until the atomic write is finished. In such doing, crash consistency is guaranteed without introducing extra writes. We then describe the specific hash index design for supporting atomic write and present the consistency algorithms.

3.4.1 Hash index design

Modern processors support 8B atomic writes natively and 16B atomic writes using cmpxchg16b instruction (with LOCK prefix) [46, 47]. However, the key size of KV stores is usually 16B [13, 48]. Directly placing the original key and the position of KV item in a hash index entry makes it impossible to apply atomic writes.

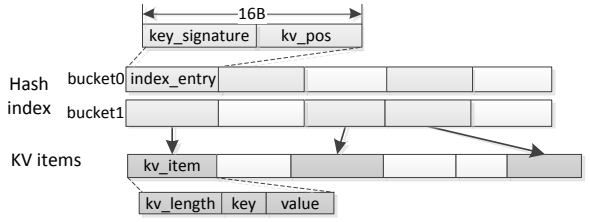


Figure 4: Hash index and key-value data layout

The position of key-values needs 48bits in modern processors. If the index entry is designed to be 8B, then the key signature can only occupy 16 bits. There exists many signature conflicts as 16 bits signature can only distinguish 65536 keys. Thus, HiKV adopts 16B index entry that can also be updated atomically. Figure 4 shows the design of the hash index and KV items. A 16B index entry consists of a 64bits key signature and a 64bits position to refer the position of KV item. A hash bucket contains multiple 16B index entries. To support varied-length key and value, each KV item stores a 32bit kv_length, key and value. Key signature may still be conflicted among different keys. Thus, HiKV checks corresponding KV item if the key signature in index entry equals to the signature of specified key.

3.4.2 Consistency algorithm

In this subsection, we present the consistency algorithms of different HiKV operations. Note that, memory writes may be reordered due to the caching of CPU or the scheduling of memory controller. HiKV uses the sequence of *sfence, clflush, sfence* instruction (referred to *persist*) to enforce the ordering and persistency of memory writes based on existing hardware [24, 27, 28, 49, 50]. The *clflush* can be replaced with the latest *CLWB* instruction [51] if the hardware supports it.

Put. Algorithm 1 presents the pseudo-code of *Put*. It first finds an empty index entry (line 1). Then the algorithm allocates free space to store the KV item (line 2). Next, it sets the KV item (line 3), and persists the KV item to NVM (line 4). At last, it performs an atomic write to set the index entry(line 5), and persists the index entry (line 6).

Algorithm 1 HiKV_PUT(op, key, value)

```
1: index_entry = find_empty_entry(key);
2: new_kv_item = alloc_space(sizeof(kv_item));
3: set new_kv_item according to key and value;
4: persist(new_kv_item, sizeof(kv_item));
5: AtomicWrite(index_entry, new_entry);
6: persist(index_entry, sizeof(index_entry));
```

Update. Algorithm 2 presents the pseudo-code of *Update*. The algorithm finds the original index entry according to the key (line 1), and uses the index entry to find the original KV item in NVM (line 2). Since HiKV adopts out-of-place update for KV item, it needs to allocate free space to store new KV item (line 3). Then, it sets the KV item, persists it, atomically updates the index entry, and persists it like *Put* (line 4-7). At last, it deallocates the space of original KV item (line 8).

Algorithm 2 HiKV_UPDATE(op, key, value)

```
1: index_entry = find_index_entry(key);
2: orig_kv_item = get_original_item(index_entry);
3: new_kv_item = alloc_space(sizeof(kv_item));
4: set new_kv_item according to key and value;
5: persist(new_kv_item, sizeof(kv_item));
6: AtomicWrite(index_entry, new_entry);
7: persist(index_entry, sizeof(index_entry));
8: free_space(orig_kv_item);
```

Delete. Algorithm 3 presents the pseudo-code of the *Delete* operation. The algorithm first finds the original index entry and KV item (line 1, 2). It invalids the index entry by setting it to 0 using an atomic write (line 3), and then persists the index entry (line 4). At last, it deallocates the space of original KV item (line 5).

Algorithm 3 HiKV_DELETE(op, key)

```
1: index_entry = find_index_entry(key);
2: orig_kv_item = get_original_item(index_entry);
3: AtomicWrite(index_entry, 0);
4: persist(index_entry, sizeof(index_entry));
5: free_space(orig_kv_item);
```

The validity of a KV item is identified by corresponding index entry. Since the index entry is atomically updated at last, crashes happened in any step of the three algorithms do not destroy consistency.

Note that, HiKV faces the challenge of memory leak when a crash occurs after allocating a free NVM space. Solving memory leak thoroughly relies on the support of underlying libraries and operating system. We leave it as our future work.

3.5 Recovery

In this section, we describe the recovery of HiKV after normal shutdown and system failure.

Recovery after a normal shutdown. On a normal shutdown, HiKV persists the B^+-Tree index in continuous NVM space. Then, HiKV saves the start address of this space to a reserved position in NVM and atomically writes a flag indicating a normal shutdown. HiKV checks the flag when it recovers the index. If the flag indicates a normal shutdown, then HiKV reads the B^+-Tree index stored in NVM and recovers it to DRAM. Otherwise, HiKV executes the following recovery.

Recovery after a system failure. In case of a system failure, HiKV must rebuild the B^+-Tree index from the consistent hash index and key-value items in NVM by only scanning all hash indexes. For each index_entry in every hash_index, if its value is not zero, the recovery thread inserts the key and the position of corresponding KV item to the B^+-Tree index. Otherwise, the index_entry is invalid.

3.6 Implementation

We implement HiKV on top of the hybrid index. HiKV utilizes the lossless hash index design in MICA [42]. A hash bucket contains multiple successive index entries. HiKV sequentially searches next index entry in the hash bucket when a hash collision occurs. Each index entry in the leaf nodes of B^+-Tree contains a whole key and the position of corresponding KV item in NVM. We implement multiple lock-free updating queues to reduce contention when serving highly concurrent requests. All backend threads poll updating queues as the cost of thread synchronization is high.

4 Evaluation

In this section, we evaluate the performance of HiKV. We first describe the experimental setup and then evaluate HiKV using micro- and macro-benchmarks.

4.1 Experimental Setup

We conduct all experiments on a server equipped with two Intel Xeon E5-2620 v4 processors. Each one running at 2.1 GHz has 8 cores, a shared 20MB last level cache. The memory size in the server is 256GB.

NVM emulation

As real NVM DIMMs are not available for us yet, we emulate NVM using the DRAM similar to prior works [49, 52, 6]. The access latency of DRAM is about 60 ns [49], and the write latency of the latest 3D-XPoint is ten times of DRAM [3]. Thus, we set the NVM write latency to 600 ns. We add extra write latency only once for each *persist* operation as described in Section 3.4.2. We add the long write latency of NVM using the x86 RDTSCP instruction. We use the instruction to read the processor timestamp counter and spin until the counter reaches the configured latency. We do not add extra read latency for NVM as it has similar read latency with DRAM [28, 31]. The impact of longer NVM read latency is evaluated in Subsection 4.6.

Workloads

We use five micro-benchmarks to evaluate the performance of single KV operations, namely *Put*, *Get*, *Update*, *Delete*, and *Scan*. The randomly generated scan count is less than 100 like YCSB [53]. For each micro-benchmark, we first warm up KV stores with 50M key-values. Then, we execute 50M operations with randomly selected key-values. All our micro-benchmarks generate KV data following the uniform distribution. We use the widely used macro-benchmark YCSB to evaluate the performance of mixed operations. We also execute 50M key-value operations. We use the default configuration of YCSB that is zipfian distribution with 99% skewness.

For both micro- and macro- benchmarks, we always use a key size of 16B, which is a typical size in production environment [13, 48]. In Facebook, over 90% value sizes of Memcached are close but less than 500B [48]. Thus, we set the value size to 256B basically.

Compared systems

We compare HiKV with the state-of-the-art NVM KV store NVStore [28] and hybrid memory KV store FPTree [30]. We do not compare HiKV with disk-based KV stores, such as RocksDB [11]. This is because HiKV is designed for byte-addressable NVM, and its I/O stack is quite different from that of RocksDB. We also do not evaluate KV stores that periodically persist data, such as Echo [54] and Masstree [39]. These KV stores cannot guarantee the consistency of every KV operation.

We re-implement NVStore and FPTree as faithfully as possible according to the descriptions in their papers. The index of NVStore is an optimized B^+-Tree, called NVTree, which keeps entries in leaf nodes unsorted to reduce NVM writes. To be fair, we place inner nodes of NVTree in the DRAM as the way HiKV uses the DRAM. FPTree also uses a variation of B^+-Tree, which adds a bitmap and fingerprints in each unsorted leaf node to accelerate searching.

A typical usage of DRAM for hybrid memory systems is using DRAM as a cache of NVM, besides placing a

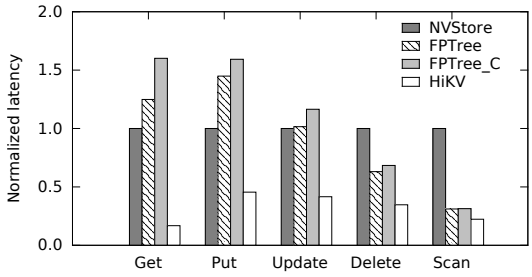


Figure 5: Normalized latency of KV ops

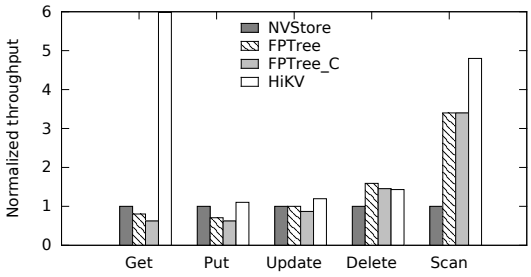


Figure 6: Normalized throughput of KV ops

part of index in DRAM. For 16B keys and 256B values, HiKV's ratio of DRAM consumption to NVM is 15.4% larger than FPTree (details in Subsection 4.7). Thus, we use the extra DRAM as a cache of FPTree, called FPTree_C in our evaluation. FPTree_C uses hash index and LRU replacement policy to manage the cache.

4.2 Single-threaded performance

We first evaluate the single-threaded performance of HiKV using micro-benchmarks. For benchmarks that only read data, including *Get* and *Scan*, all four KV stores use one thread. Note that, HiKV is designed to adopt serving threads accompanied with backend threads to operate the B^+-Tree index when serving write requests. Thus, for *Put*, *Update*, and *Delete*, HiKV is configured to use one serving thread and one backend thread. For fair comparison, both NVStore, FPTree, and FPTree_C are configured with two threads.

4.2.1 Latency reduction

Figure 5 shows the latency reduction of HiKV. For *Get*, HiKV can reduce latency by 83.2% and 86.6% than NVStore and FPTree, respectively. HiKV only needs to lookup the fast hash index. However, both NVStore and FPTree not only need to lookup the tree index, but also need to sequentially lookup a leaf node as keys in the leaf node are unsorted. For *Put*/*Update*/*Delete*, HiKV can reduce latency by 54.5%/58.4%/65.3% than NVStore, and 68.8%/59.1%/45.0% than FPTree, respectively. This is because searching the hash index is fast and HiKV uses asynchronous mechanism to hide the latency of B^+-Tree index. For *Put* and *Update*, FPTree needs to persist data three times (bitmap, fingerprints and key-value), while NVStore only needs to persist data twice (key-value and leaf.number). As a result, the *Put* and *Update* latencies of FPTree are higher than those of NVStore. For *Delete*, HiKV only needs to invalid the corresponding index_entry and persist it to NVM. However, NVStore needs to insert the key-value with an invalid flag and update the leaf_number, which persists data to NVM twice. Although FPTree only needs to invalid bitmap and persist once for *Delete*, its latency is still larger than that of HiKV due to inefficient searching of tree index.

For *Scan*, HiKV can reduce latency by 77.7% and 28.3% than NVStore and FPTree, respectively. When putting a key-value, NVStore does not check whether the key-value has existed in the leaf node. As a result, it must check whether a key-value is valid or not when scanning a leaf node. Thus, the *Scan* latency of NVStore is apparently larger than that of HiKV. FPTree uses a bitmap per leaf node to identify the validity of key-value entries in the leaf node. Thus, the *Scan* latency of FPTree is lower than that of NVStore, while it is still larger than that of HiKV.

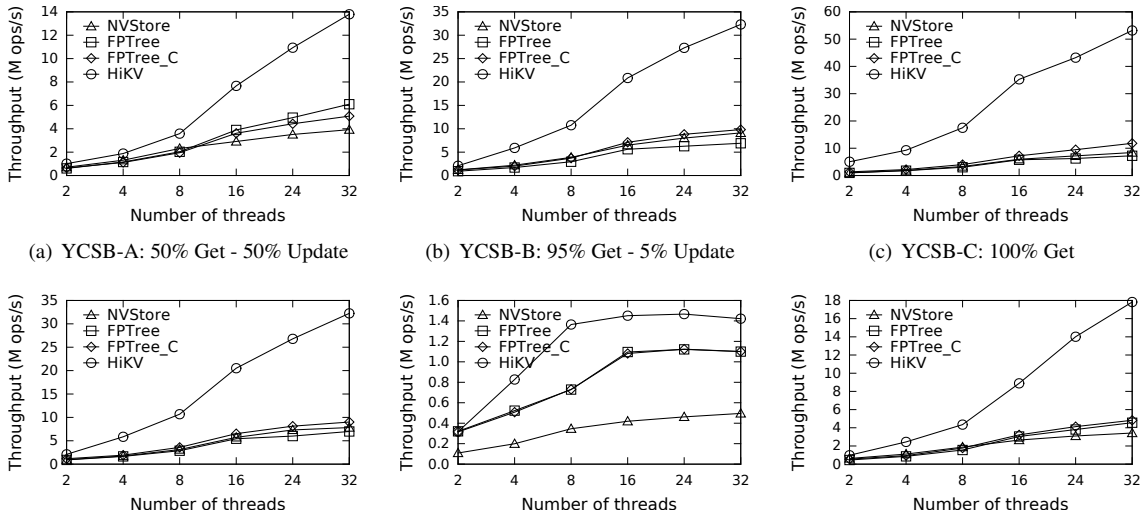
FPTree_C performs worse than FPTree for single-key operations. This is because the micro-benchmarks have uniform distribution, which results in low cache hit ratio. FPTree_C incurs extra performance overhead for the cache replacement.

4.2.2 Throughput improvement

Figure 6 shows the throughput improvement of HiKV. HiKV can improve throughput by 5.0x/3.8x than NVStore, and 6.4x/41.2% than FPTree for *Get*/*Scan*, respectively. For *Put*/*Update*, HiKV outperforms NVStore and FPTree by 10.4%/19.6%, and 55.9%/19.6%, respectively. The *Delete* throughput of HiKV is 43.2% higher than that of NVStore, and 10.0% lower than that of FPTree. The throughput improvement of HiKV is lower in *Put*/*Update*/*Delete* than in *Get*/*Scan*. This is because NVStore and FPTree use two threads to run these write requests, while HiKV only uses one serving thread. For read requests, these three KV stores use one thread. FPTree_C achieves lower throughput than FPTree due to the overhead of DRAM cache management.

4.3 Scalability

We then evaluate the scalability of HiKV using the macro-benchmark YCSB. We do not use the original YCSB framework with client-server mode due to its long latency of network stack. Here, we use a local YCSB workload generator following the default YCSB configurations like MICA [42]. HiKV dynamically adapts the number of serving threads and back-



(a) YCSB-A: 50% Get - 50% Update

(b) YCSB-B: 95% Get - 5% Update

(c) YCSB-C: 100% Get

(d) YCSB-D: 95% Get - 5% Put

(e) YCSB-E: 95% Scan - 5% Put

(f) YCSB-F: 50% Get - 50% ReadModify-Write

Figure 7: Scalability on YCSB workloads

end threads according to different workloads. To be fair, we configure NVStore, FPTree, and FPTree_C to use the same total number of threads as HiKV in all evaluations. We run these benchmarks at most 32 threads as the server we used has 16 cores.

Figure 7 presents the throughputs of YCSB workloads as the number of threads varies from 2 to 32. The performance of HiKV with 32 threads is increased by a factor of 13.6/15.5/10.5/15.4/4.3/18.3 for YCSB-A/B/C/D/E/F against the two-threaded execution, respectively. For the same scalability evaluation, the scaling factors for NVStore, FPTree, and FPTree_C are 5.5/7.6/8.2/8.2/4.6/5.5, and 10.0/7.5/7.5/7.7/3.4/10.1, and 7.9/8.2/8.8/7.8/3.5/8.8, respectively. In summary, HiKV achieves better scalability than NVStore and FPTree. The *Get* ratio is 95%, 95%, and 75% for YCSB-B, YCSB-D, and YCSB-F, respectively. HiKV provides more than 20 serving threads with 32-threaded execution due to the dynamic threads adaption. However, HiKV only has one serving thread with 2-threaded execution. As a result, HiKV executed with 32 threads can improve throughput by large than 15x than 2-threaded execution for YCSB-B/D/F.

With 32-threaded execution, HiKV outperforms NVStore by 1.7x to 5.3x, FPTree by 24.2% to 6.3x, and FPTree_C by 24.1% to 3.5x. For read-intensive and skewed workloads, such as YCSB-B/C, FPTree_C performs better than FPTree for as the cache hit ratio is high. For YCSB-E, HiKV can scale to 8 threads almost linearly and keeps stable with more threads. This is because HiKV must lock all updating queues temporally before serving *Scan*, which would block the *Put* of other threads. NVStore, FPTree and FPTree_C can scale to 16 threads for YCSB-E. Even so, HiKV still improves throughput by 1.7x, 24.2%, and 24.1% than NVStore, FPTree and FPTree_C, respectively.

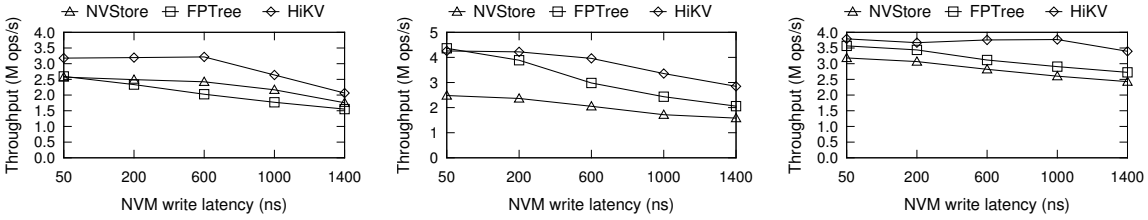
4.4 Sensitivity analysis

In this section, we conduct sensitivity analysis to HiKV considering NVM write latency and workload dataset size. We use 16 threads for all the experiments.

4.4.1 Sensitivity to NVM write latency

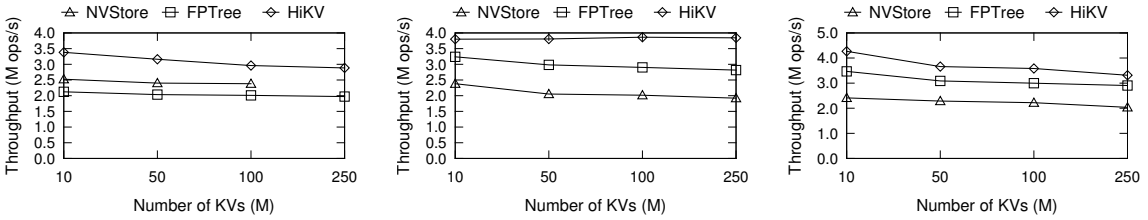
The write latencies are different among various NVM devices. Thus, we evaluate the impact of NVM write latency on the performance. Figure 8 shows the throughput results when we vary NVM write latency from 50 ns to 1400 ns. The *Get* and *Scan* performance have no relation with the write latency. Thus, we only show the results of *Put*, *Update*, and *Delete*. We do not show the results of FPTree_C as it performs worse than FPTree for uniform distributed workloads.

We find that the throughput decreases as NVM write latency increases for NVStore and FPTree. This is due to the increase of persist latency. For *Delete*, the throughput of HiKV remains stable when the write latency is lower than 1400 ns. This is because the concurrent deletion latency of B^+-Tree index is still longer than that of the hash index even though the write latency increases to 1000 ns. Compared to NVStore and FPTree, HiKV still improves the throughput of *Delete* by 17.6%/80.0%/39.9%, and 32.9%/38.4%/24.6% for *Put*/*Update*/*Delete*, respectively even if the write latency of NVM reaches 1400 ns.



(a) Put (b) Update (c) Delete

Figure 8: Impact of NVM write latency



(a) Put (b) Update (c) Delete

Figure 9: Impact of dataset size

4.4.2 Sensitivity to dataset size

HiKV adopts a global B$^+$-Tree index to support *Scan*. A doubt is whether the B$^+$-Tree index limits the scalability of HiKV as the dataset size increases. Figure 9 shows the throughput as the number of key-values increases from 10M to 250M. The *Put* throughput of NVStore is not available for 250M key-values as they run out of our server memory due to the sharply increased size of the NVTree index. The total number of key-values is 500M for *Put* as we first warm up with 250M key-values.

The throughput of HiKV remains unchanged for *Update*, while the throughput of NVStore and FPTree decreases by 19.3% and 13.1%, respectively. When the number of key-values increases 25 times, the throughput of HiKV, NVStore, and FPTree decreases by 14.6% /22.4%, NA/15.6%, and 7.2%/16.3% for *Put* and *Delete*, respectively. The performance degradation is due to the increased searching latency with increased dataset size. The update throughput of HiKV is determined by serving threads under such configuration. Thus, we can conclude that the global B$^+$-Tree index does not limit the scalability compared to NVStore and FPTree.

4.5 Performance breakdown

In this section, we first analyze the effectiveness of asynchronous updates, differential concurrency, and ordered-write consistency of HiKV. HiKV_sync updates the hash index and B$^+$-Tree synchronously within one thread. HiKV_par adopts partitioning-based concurrency control for B$^+$-Tree index, which has ordered multi-B$^+$-Tree indexes. HiKV_wal uses the traditional Write-Ahead Log to guarantee consistency.

Figure 10 shows the average latency of *Put* as the NVM write latency increases from 50 ns to 1400 ns. Compared with HiKV_sync, HiKV can reduce latency by 46.7% to 57.8%. This is due to the asynchronous updates of HiKV that the critical path only contains operating the hash index. HiKV can reduce latency by 11.2% to 17.4% compared to HiKV_par. The performance degradation of HiKV_par is caused by the two reasons. First, migrating index entries among B$^+$-Tree indexes blocks normal put operations. Second, the migration thread preempts CPU resources in 16-threaded execution. HiKV_wal stores key and value position (index_entry) in hash index. To guarantee consistency, HiKV_wal first writes key-value in log area, then write value to NVM and writes the index_entry in hash index for *Put*. Writing value and index_entry in hash table without logging will result in inconsistency. This is because the index_entry and value in HiKV_wal can not be update atomically. We can find that HiKV_wal needs to persist data to NVM three times although HiKV_wal does not need to guarantee the order of writing value and index_entry to NVM. However, HiKV only needs to persist data to NVM twice due to the order-writing. The evaluation result shows that HiKV can reduce latency than HiKV_wal by up to 27.4% when the NVM write latency reaches 1400 ns.

Secondly, we evaluate the effectiveness of dynamic threads adaption in HiKV. We first warmup 50M key-values, then we execute back-to-back YCSB-

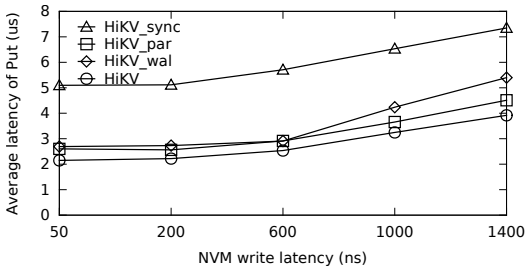


Figure 10: Effectiveness of asynchronous updates, differential concurrency, and order-writing consistency

A/B/C/D/E/F. The percentages of *Put/Get/Update/Scan* are varied in these workloads. Each workload is executed by 60 sec. The total number of threads is 16. HiKV-8-8 and HiKV-12-4 represents executing with static 8 serving threads and 8 backend threads, and 12 serving threads and 4 backend threads, respectively.

Figure 11 shows the throughputs as the workload changes from YCSB-A to YCSB-F. HiKV can achieve the highest throughput except for YCSB-A. For YCSB-B/C/D/E/F, HiKV outperforms HiKV-8-8 and HiKV-12-4 by 10.5% to 1.0x and 10.4% to 37.5%, respectively. For YCSB-A, the throughput of HiKV is same throughput with HiKV-8-8, and slightly lower than HiKV-12-4 by 1.6%. HiKV dynamically adapts the number of serving threads and backend threads, such as 8 and 8 for YCSB-A, 13 and 3 for YCSB-B, 9 and 7 for YCSB-F. For read-intensive workloads, increasing the number of serving threads can improve throughput of HiKV.

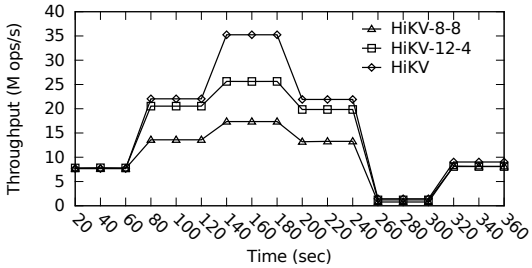


Figure 11: Effectiveness of dynamic threads adaption. *0-60s:YCSB-A. 60-120s:YCSB-B. 120-180s:YCSB-C. 180-240s:YCSB-D. 240-300s:YCSB-E. 300-360s:YCSB-F.*

4.6 Impact of NVM read latency

A few researches indicate that the read of NVM is longer than that of DRAM [47, 55]. Thus, we evaluate the impact of NVM read latency on system performance. We emulate the longer read latency similar to emulating write latency. We set the NVM read latency to 120 ns, which is twice as that of DRAM [55].

Figure 12 shows that the average serving latency increases as NVM read latency does. This is because HiKV spends more time to search hash index. However, NVStore and FPTree also takes more time when searching unsorted leaf nodes and splitting/merging leaf nodes. Thus, HiKV can still apparently reduce latency than NVStore and FPTree. For example, HiKV can reduce latency by 80.0%/61.8% than NVStore, and 82.3%/13.0% than FPTree for *Get*/*Scan* with doubled read latency, respectively.

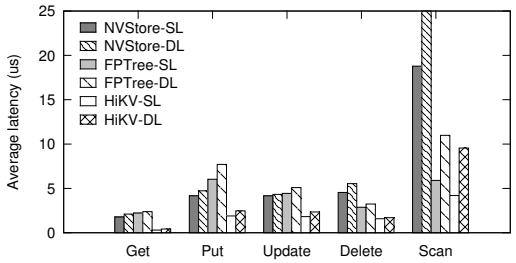


Figure 12: Impact of NVM read latency. (*SL* and *DL* represent same latency and doubled latency as DRAM.)

4.7 Memory consumption

Figure 13 shows the DRAM and NVM consumptions after randomly putting 50M key-values to different KV stores. The value size varies from 64B to 1KB. The curves show the ratio of DRAM consumption to NVM consumption. Since DRAM is used to store the indexes of key-values, the DRAM consumptions are related to the number of key-values and keep constant with varied value sizes for both KV stores. On the contrary, in both KV stores, NVM is used to store data items and its consumption increases as the value size increases.

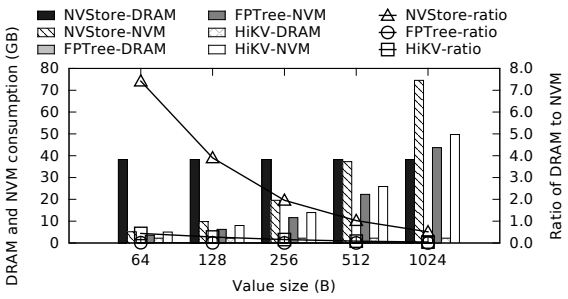


Figure 13: DRAM consumption

We observe that NVStore consumes DRAM as high as 38.27 GB. This is mainly because NVStore creates one parent-leaf-node per leaf node when rebuilding the contiguous inner nodes of NVTree. The index size of NVTree increases exponentially as the tree height increases. HiKV always consumes 2.21 GB DRAM to store B$^+$-Tree index, which is larger than FPTree. This

is because the fine-grained B$^+$-Tree of HiKV must index every unsorted key-value in NVM, while FPTree only stores its inner nodes in DRAM. However, the HiKV-ratio decreases from 40% to 4% as the value size increases from 64B to 1KB. For 256B value, the HiKV-ratio is 15.8%. Our evaluation shows that even though the extra DRAM space is used as a cache of FPTree (namely FPTree_C), HiKV still achieves higher performance than FPTree_C. Reducing the DRAM consumption of B$^+$-Tree, such as migrating part of leaf nodes to NVM, is our future work.

4.8 Recovery time

We finally evaluate the recovery performance of HiKV, NVStore, and FPTree. NVStore and FPTree takes 11.03s and 1.74s to recover 50M key-values, respectively. NVStore takes more time than FPTree as it allocates much larger contiguous inner nodes for tree index and insert keys more randomly than FPTree. Since the hash index is unsorted, HiKV needs to read valid index_entry in NVM and insert corresponding key and key-value position to the B$^+$-Tree index one by one. Thus, HiKV takes 88.24s to recover 50M key-values with one thread. However, increasing recovery threads allows to reduce the recovery time. For instance, HiKV takes 6.28s to recover 50M key-values with 16 threads.

5 Related Work

In this section, we discuss related works from three aspects: indexing structure, concurrency control, and NVM consistency guaranteeing.

Indexing Structure. Several distributed KV stores, such as Cassandra [43], Megastore [44], and SLIK [45], construct multiple indexes for multi-key-value data, such as secondary index for non primary key query. However, HiKV constructs a hybrid index according to a single key, and focuses on reducing the latency of updating hybrid index. SILT [18] and dual-stage index [19] construct multiple indexes to reduce DRAM consumption of indexes. These two techniques are orthogonal to HiKV to reduce the DRAM consumption of B$^+$-Tree.

NVM, especially PCM, suffers from limited write endurance. Thus, a number of research efforts are made to optimize the indexing structure for NVM to reduce writes to NVM [23, 25, 56, 57]. For example, Chen et al. [23] propose the unsorted leaf nodes of B$^+$-Tree to writes caused by sorting. Instead of focusing on reducing NVM writes, HiKV mainly optimizes indexing structure to support rich KV operations.

Concurrency Control. Concurrency control for multi-core processor has been widely studied in KV stores. Echo [54] and NVStore [28] use the MVCC for concurrency control. Chronos [58] and MICA [42] uses partitioning for concurrency control of hash tables. PALM [59] is lock-free concurrent B$^+$-Tree. FPTree adopts HTM to handle the concurrency of inner nodes, and fine-grained locks for the concurrency access of leaf nodes[30]. HiKV adopts similar techniques according to the features of hybrid index, which are partitioning for hash tables and HTM for B$^+$-Tree index.

NVM consistency guaranteeing. Recent research works propose techniques to reduce the cost of consistency guaranteeing. A few research works [60, 61, 62] use the differential logging [63] to only record modified bytes of a block on journal to reduce NVM writes. However, differential logging cannot avoid twice writes. Several works propose a combination of multiple techniques to reduce consistency cost according to data granularity. Atomic-write is used to update file system metadata, whose granularity is usually small such as 8B or 16B [5, 47, 64]. For large-granularity data, write-ahead logging and copy-on-write are used [5, 47]. NVStore [28], FPTree [30] also use ordered-write to guarantee consistency. However, HiKV adopts ordered-write accompanied with atomic-write to hash index, which can always achieve the minimum count of persists for different KV write operations.

6 Conclusion

Persistent key-value stores are widely deployed in real-world applications. Hybrid memory consisting of DRAM and NVM allows storage systems to persist data directly in the memory. Building KV stores towards hybrid memory can exploit its performance characteristic. Supporting rich KV operations (*Put/Get/Update/Delete/Scan*) efficiently is highly required by today's applications. However, either hash index or B$^+$-Tree index employed by existing KV stores cannot efficiently support all these operations. In this paper, we propose hybrid index to adopt a hash index in NVM for fast searching and directly persisting, and a B$^+$-Tree index in DRAM for fast updating and supporting range scan. On top of the hybrid index, we build HiKV, a hybrid index based key-value store with the well-performed scalability and crash consistency guaranteeing. Extensive experiments show that HiKV outperforms the state-of-the-art NVM-based KV stores.

7 Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Donald E. Porter, for their helpful comments. We also thank Ismail Oukid and Jun Yang for their help in figuring out the details of FPTree and NVStore, respectively. We thank Wenlong Ma for useful discussions. This work is supported by National Key Research and Development Program of China under grant No. 2016YFB1000302 , National Science Foundation of China under grant No. 61502448 and No. 61379042.

References

[1] H. Wong, S. Raoux, S. Kim, J. Liang, J. Reifenberg, B. Rajendran, M. Asheghi, and K. Goodson, "Phase change memory," *Proceedings of the IEEE*, vol. 98, pp. 2201–2227, 2010.

[2] I. Baek, M. Lee, S. Seo, M. Lee, D. Seo, D.-S. Suh, J. Park, S. Park, H. Kim, and I. Yoo, "Highly scalable nonvolatile resistive memory using simple binary oxide driven by asymmetric unipolar voltage pulses," in *2004 IEEE International on Electron Devices Meeting*, IEDM'04, pp. 587–590, 2004.

[3] "Intel and Micron produce breakthrough memory technology." `https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough\-memory-technology/`.

[4] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pp. 24–33, 2009.

[5] X. Jian and S. Steven, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies*, FAST'16, pp. 323–338, 2016.

[6] J. Ou, J. Shu, and Y. Lu, "A high performance file system for non-volatile main memory," in *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pp. 12:1–12:16, 2016.

[7] 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 *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '06, pp. 15–15, 2006.

[8] S. Ghemawat and J. Dean, "LevelDB." `https://leveldb.org`.

[9] 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 *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pp. 205–220, 2007.

[10] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah, "Serving large-scale batch computed data with project voldemort," in *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, pp. 18–18, 2012.

[11] Facebook, "RocksDB." `https://rocksdb.org`.

[12] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in haystack: Facebook's photo storage.," in *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, OSDI'10, pp. 1–8, 2010.

[13] 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 *Proceedings of the 31st Symposium on Mass Storage Systems and Technologies*, MSST'15, pp. 1–14, IEEE, 2015.

[14] P. Shetty, R. P. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok, "Building workload-independent storage with VT-trees," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, pp. 17–30, 2013.

[15] L. Marmol, S. Sundararaman, N. Talagala, and R. Rangaswami, "NVMKV: A scalable, lightweight, ftl-aware key-value store," in *Proceedings of the 2015 USENIX Annual Technical Conference*, USENIX ATC'15, pp. 207–219, 2015.

[16] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in SSD-conscious storage," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies*, FAST'16, pp. 133–148, Feb. 2016.

[17] B. Debnath, S. Sengupta, and J. Li, "Flashstore: High throughput persistent key-value store," *Proceedings of the VLDB Endowment*, vol. 3, pp. 1414–1425, Sept. 2010.

[18] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "SILT: A memory-efficient, high-performance key-value store," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pp. 1–13, 2011.

[19] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen, "Reducing the storage overhead of main-memory oltp databases with hybrid indexes," in *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pp. 1567–1581, 2016.

[20] J. Levandoski, D. Lomet, and S. Sengupta, "The Bw-Tree: A b-tree for new hardware platforms," in *Proceedings of the IEEE 29th International Conference on Data Engineering*, ICDE'13, pp. 302–313, 2013.

[21] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar, "Scaling concurrent log-structured data stores," in *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pp. 32:1–32:14, 2015.

[22] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong, "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*, EuroSys '14, pp. 16:1–16:14, 2014.

[23] S. Chen, P. B. Gibbons, and S. Nath, "Rethinking database algorithms for phase change memory," in *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, CIDR'11, pp. 21–31, 2011.

[24] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, pp. 5–5, 2011.

[25] J. Ni, W. Hu, G. Li, K. Tan, and D. Sun, "Bp-tree: A predictive b+-tree for reducing writes on phase change memory," *IEEE Transactions on Knowledge and Data Engineering*, vol. PP, no. 99, pp. 1–1, 2014.

[26] P. Chi, W.-C. Lee, and Y. Xie, "Making b+-tree efficient in pcm-based main memory," in *Proceedings of the 2014 International Symposium on Low Power Electronics and Design*, ISLPED '14, pp. 69–74, 2014.

[27] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," *Proceedings of the VLDB Endowment*, vol. 8, pp. 786–797, Feb. 2015.

[28] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "NV-Tree: Reducing consistency cost for nvm-based single level systems," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pp. 167–181, 2015.

[29] G. S. Choi, B. W. On, and I. Lee, "Pb+-tree: Pcm-aware b+-tree," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 9, pp. 2466–2479, 2015.

[30] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "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*, SIGMOD '16, pp. 371–386, 2016.

[31] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pp. 2–13, 2009.

[32] K. Suzuki and S. Swanson, "The non-volatile memory technology database (NVMDB)," Tech. Rep. CS2015-1011, Department of Computer Science & Engineering, University of California, San Diego, 2015.

[33] M. F. Chang, J. J. Wu, T. F. Chien, Y. C. Liu, T. C. Yang, W. C. Shen, Y. C. King, C. J. Lin, K. F. Lin, Y. D. Chih, S. Natarajan, and J. Chang, "19.4 embedded 1mb reram in 28nm cmos with 0.27-to-1v read using swing-sample-and-couple sense amplifier and self-boost-write-termination scheme," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, ISSCC'14, pp. 332–333, Feb 2014.

[34] Micron, "SLC NAND flash products." `http://www.micron.com/products/nand-flash/slc-nand#fullPart,Dec.2014`.

[35] "MyRocks: A space- and write-optimized mysql database." `https://code.facebook.com/posts/190251048047090/myrocks-a-space-and-write-optimized-mysql-database/`.

[36] K. Ren and G. Gibson, "TABLEFS: Enhancing metadata efficiency in the local file system," in *Proceedings of the 2013 USENIX Annual Technical Conference*, USENIX ATC'13, pp. 145–156, 2013.

[37] 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 *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pp. 307–320, 2006.

[38] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion," in *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'14, pp. 237–248, 2014.

[39] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pp. 183–196, 2012.

[40] 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, pp. 18–32, 2013.

[41] "Memcached." `https://memcached.org`.

[42] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A holistic approach to fast in-memory key-value storage," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'14, pp. 429–444, 2014.

[43] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[44] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *Proceedings of the 5th Conference on Innovative Data system Research*, CIDR'11, pp. 223–234, 2011.

[45] A. Kejriwal, A. Gopalan, A. Gupta, Z. Jia, S. Yang, and J. Ousterhout, "SLIK: Scalable low-latency indexes for a key-value store," in *Proceedings of the 2016 USENIX Annual Technical Conference*, USENIX ATC'16, pp. 57–70, 2016.

[46] "Intel64 software developers manual (vol 2, ch 3.2)," 2013.

[47] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pp. 15:1–15:15, ACM, 2014.

[48] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "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*, SIGMETRICS '12, pp. 53–64, 2012.

[49] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: lightweight persistent memory," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pp. 91–104, 2011.

[50] X. Wu and A. L. N. Reddy, "Scmfs: a file system for storage class memory," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pp. 39:1–39:11, 2011.

[51] "Intel architecture instruction set extensions programming reference." `https://software.intel.`

com/sites/default/files/managed/69/78/319433-025.pdf.

[52] J. Huang, K. Schwan, and M. K. Qureshi, "Nvram-aware logging in transaction systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 4, pp. 389–400, 2014.

[53] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pp. 143–154, 2010.

[54] K. A. Bailey, P. Hornyack, L. Ceze, S. D. Gribble, and H. M. Levy, "Exploring storage class memory with key value stores," in *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '13, pp. 4:1–4:8, 2013.

[55] Y. Zhang and S. Swanson, "A study of application performance with non-volatile main memory," in *Proceedings of the 31st Symposium on Mass Storage Systems and Technologies*, MSST'15, pp. 1–10, IEEE, 2015.

[56] B. Debnath, A. Haghdoost, A. Kadav, M. G. Khatib, and C. Ungureanu, "Revisiting hash table design for phase change memory," *ACM SIGOPS Operating System Review*, vol. 49, pp. 18–26, Jan 2016.

[57] P. Zuo and Y. Hua, "A write-friendly hashing scheme for non-volatile memory systems," in *Proceedings of the 33st Symposium on Mass Storage Systems and Technologies*, MSST'17, pp. 1–10, 2017.

[58] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat, "Chronos: Predictable low latency for data center applications," in *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pp. 9:1–9:14, 2012.

[59] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey, "PALM: Parallel architecture-friendly latch-free modifications to b+ trees on many-core processors," *Procedings of the VLDB Endowment*, vol. 4, no. 11, pp. 795–806, 2011.

[60] E. Lee, S. Yoo, J.-E. Jang, and H. Bahn, "Shortcut-jfs: A write efficient journaling file system for phase change memory," in *Proceedings of the IEEE 28th Symposium on Mass Storage Systems and Technologies*, MSST'12, pp. 1–6, 2012.

[61] J. Chen, Q. Wei, C. Chen, and L. Wu, "FSMAC: A file system metadata accelerator with non-volatile memory," in *Proceedings of the IEEE 29th Symposium on Mass Storage Systems and Technologies*, MSST'13, pp. 1–11, IEEE, 2013.

[62] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won, "Nvwal: Exploiting nvram in write-ahead logging," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pp. 385–398, 2016.

[63] J. Lee, K. Kim, and S. K. Cha, "Differential logging: A commutative and associative logging scheme for highly parallel main memory database," in *Proceedings of the 17th International Conference on Data Engineering*, ICDE'01, pp. 173–182, IEEE, 2001.

[64] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *Proceedings of the Twenty-Second ACM Symposium on Operating Systems Principles*, SOSP '09, pp. 133–146, 2009.

TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores

Oana Balmau *EPFL*
Diego Didona *EPFL*
Rachid Guerraoui *EPFL*
Willy Zwaenepoel *EPFL*
Huapeng Yuan *Nutanix*
Aashray Arora *Nutanix*
Karan Gupta *Nutanix*
Pavan Konka *Nutanix*

Abstract

We present TRIAD, a new persistent key-value (KV) store based on Log-Structured Merge (LSM) trees. TRIAD improves LSM KV throughput by reducing the write amplification arising in the maintenance of the LSM tree structure. Although occurring in the background, write amplification consumes significant CPU and I/O resources. By reducing write amplification, TRIAD allows these resources to be used instead to improve user-facing throughput.

TRIAD uses a holistic combination of three techniques. At the LSM memory component level, TRIAD leverages skew in data popularity to avoid frequent I/O operations on the most popular keys. At the storage level, TRIAD amortizes management costs by deferring and batching multiple I/O operations. At the commit log level, TRIAD avoids duplicate writes to storage.

We implement TRIAD as an extension of Facebook's RocksDB and evaluate it with production and synthetic workloads. With these workloads, TRIAD yields up to 193% improvement in throughput. It reduces write amplification by a factor of up to 4x, and decreases the amount of I/O by an order of magnitude.

1 Introduction

Key-value (KV) stores [1, 3, 4, 12, 29, 35, 38, 39, 47] are nowadays a widespread solution for handling large-scale data in cloud-based applications. They have several advantages over traditional DBMSs, including simplicity, scalability, and high throughput. KV store applications include, among others, messaging [12, 18], online shopping [25], search indexing [22] and advertising [12, 22]. At Nutanix, we use KV stores for storing the metadata for our core enterprise platform, which serves thousands of customers with petabytes of storage capacity [21].

KV store systems are available for workloads that fit entirely in memory (e.g., Mica [36], Redis [3], and Memcached [2]), as well as for workloads that require persistent storage (e.g., LevelDB [4], RocksDB [12]). Log-Structured Merge trees (LSMs) [41, 40] are a popular design choice for the latter category. LSMs achieve high write throughput at the expense of a small decrease in read throughput. They are today employed in a wide range of KV stores such as LevelDB [4], RocksDB [12], Cassandra [1], cLSM [29], and bLSM [44].

Broadly speaking, LSMs are organized in two components: a *memory component* and a *disk component*. The memory component seeks to absorb updates. For applications that do not tolerate data loss in the case of failure, the updates may be temporarily backed up in a *commit log* stored on disk. When the memory component becomes full, it is flushed to persistent storage, and a new one is installed. The disk component is organized into levels, each level containing a number of sorted files, called *SSTables*. The levels closer to the memory component hold the fresher information. When level L_i is full, one or more selected files from level L_i are compacted into files at level L_{i+1}, discarding stale values. This compaction operation occurs in the background.

Compaction and flushing are key operations, responsible for maintaining the LSM structure and its properties. Unfortunately, they take up a significant amount of the available resources. For instance, for our production workloads at Nutanix, our measurements indicate that, at peak times, compaction can consume up to 45% of the CPU. Moreover, per cluster, an average of 2.5 hours per day is spent on compaction operations for the maps storing the metadata. Clearly, compaction and flushing pose an important performance challenge, even though they occur outside the critical path of user-facing operations.

We propose three new complementary techniques to close this gap. Our techniques reduce both the time and space taken by the compaction and flushing operations, leading to increased throughput. The first technique decreases compaction overhead for skewed workloads. We keep KV pairs that are updated frequently (i.e., hot en-

tries) in the memory component, and we only flush the cold entries. This separation eliminates frequent compactions triggered by different versions of the same hot entry. The main idea of the second technique is to defer file compaction until the overlap between files becomes large enough, so as to merge a high number of duplicate keys. Finally, our third technique avoids flushing the memory component altogether, by changing the role the commit logs play in LSMs and using them in a manner similar to SSTables.

Combined, our three techniques form TRIAD, a new LSM KV store we build on top of RocksDB. We extensively compare TRIAD against the original version of RocksDB on various synthetic workloads, with a focus on skewed workloads, as well as on Nutanix production workloads. TRIAD achieves up to 193% higher throughput than RocksDB. This improvement is the result of an order of magnitude decrease in I/O due to compaction and flushing, up to 4x lower write amplification and 77% less time spent compacting and flushing, on average.

To summarize, this paper makes the following key contributions: (1) the design of TRIAD, a system combining three complementary techniques for reducing compaction work in LSMs, each interesting in its own right, (2) a publicly available implementation of TRIAD as an extension of RocksDB, one of the most popular state-of-the-art LSM KV stores, and (3) an evaluation of its benefits in comparison to RocksDB.

Roadmap. The rest of the paper is structured as follows. Section 2 gives an overview of the LSM tree. Section 3 presents the background I/O overheads in LSM KV stores. Section 4 presents our three techniques for reducing the impact of the compaction and flushing operations on performance. Section 5 describes our evaluation results. Section 6 discusses related work. Section 7 concludes the paper.

2 Background on LSM

We provide an overview of the LSM structure, its user-facing operations and the flushing and compaction processes that take place in the background.

LSM Structure. The high-level view of a typical LSM-based KV store is shown in Figure 1. The system has three main components, which we briefly describe.

▷*Memory Component*. The memory component C_m is a sorted data structure residing in main memory. Its purpose is to temporarily absorb the updates performed on the KV store. The size of the memory component is typically small, ranging from a few MBs to tens of MBs. When the memory component fills up, it is replaced by a new, empty component. The old memory component is then flushed as is to level 0 (L_0) of the LSM disk component. L_0 is a special level of the disk component hierarchy, described below.

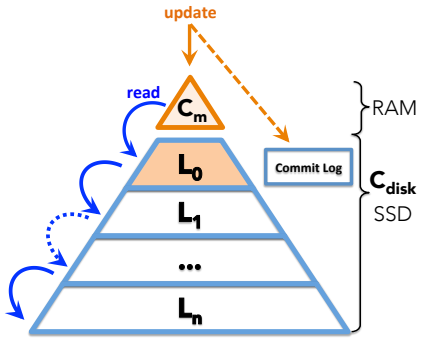


Figure 1: High-level view of a typical LSM KV store.

▷*Disk Component*. The disk component C_{disk} is structured into multiple levels (L_0, L_1, ...), with increasing sizes. Each level contains multiple sorted files, called *SSTables*. The memory component C_m is flushed to the first level, L_0. Because of this, the SSTables in L_0 have overlapping key ranges. SSTables on levels L_i ($i \neq 0$) have disjoint key ranges. The choice of the number of levels in C_{disk} is an interesting aspect of the LSM structure. From a correctness perspective, it would suffice to have only two levels on disk: one to flush memory components and one in which we compact. However, there is an I/O disadvantage to this approach. When we merge L_0 SSTables into L_1, we identify all the L_1 SSTables that have overlapping key ranges with the L_0 SSTable that is being compacted (the SSTables from L_0 cover the entire key range, because they are directly flushed from memory). If L_1 files are large, then fewer files would have overlapping key ranges, but we would have to re-write large files, leading to overhead in the compaction work and a penalty in terms of of memory use. If L_1 files are small, then a large number of files would have overlapping key ranges with the L_0 file. These files would need to be opened and rewritten, creating large overhead in the compaction work. The leveled structure allows LSMs to amortize the compaction work, as the updates trickle down the levels.

▷*Commit Log*. The commit log is a file residing on disk. Its purpose is to temporarily log the updates that are made to C_m (in small batches), if the application requires that the data is not lost in case of a failure. All updates performed on C_m are also appended to the commit log. Typically, the size of the commit log is kept small in order to provide fast recovery in case the operations need to be replayed to recover from a failure. A typical value for the size of the commit log is on the order of hundreds of MB.

User-facing operations. The main user-facing operations in LSM-based KV stores are reads ($Get(k)$) and

updates ($Update(k,v)$). $Update(k,v)$ associates value v to key k. Updates are absorbed in C_m and possibly appended to the commit log. Hence, LSM KV stores achieve high write throughput. $Get(k)$ returns the most recent value of k. As illustrated in Figure 1, the read first goes to C_m; if k is not found in C_m, the read continues to L_0, checking all the files. If k is not found in L_0, the read goes to L_1,... L_n, until k is found. Apart from L_0, only one file is checked for the rest of C_{disk}'s levels, because of the non-overlapping key ranges property.

Flushing. LSM KV stores have two main background operations: flushing and compaction. Flushing is the operation that writes C_m to L_0, once C_m becomes full. In case a commit log is used, the flush can also be triggered by the commit log getting full, even if there is still room in the memory component.

Compaction. Compaction is the background operation that merges files in disk component L_i into files with overlapping key ranges in disk component L_{i+1}, discarding older values in the process. Leveled compaction is a popular strategy for compaction in LSM KV stores [5, 14]. When the size of L_i exceeds its target size, a file F in L_i is picked and merged into the files from L_{i+1} that have overlapping key ranges with F, in a way similar to a merge sort. Therefore, in the case of leveled LSM trees, each KV pair might be eventually propagated down to the component on the last level. Hence, some KV pairs could be rewritten once for every level during compaction. RocksDB and TRIAD employ leveled compaction. The techniques proposed by TRIAD could, however, easily be adapted to size-tiered [22] approaches.

3 Motivation

Despite I/O operations not being in the critical path of user-facing operations, flushing, logging and compaction still consume computational resources. The amount of CPU cycles spent to coordinate these operations translates into a commensurate amount of processing power that cannot be used to serve the user-generated workload. Hence, the frequency and the length of the I/O operations have a significant impact on the final performance perceived by the user.

We provide experimental evidence of this claim by measuring the extent of the performance reduction due to I/O operations. We consider two workloads that exhibit different levels of skew in the data popularity (skewed/uniform) and two read/write mixes (write dominated and balanced).

We run these workloads on RocksDB and on a version of RocksDB in which we disable background I/O operations (i.e., flushing and compaction; logging was enabled for both experiments). We pin all of the system activity

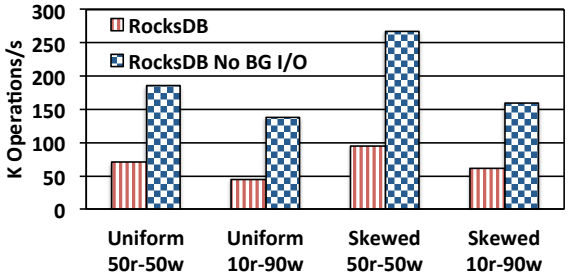


Figure 2: Background I/O impact on throughput.

(i.e., 8 worker threads and all threads created by the KV store) to 8 cores. The LSM structures of the two systems are pre-populated with an identical value for every key accessed during the experiment. This ensures that every read operation can be served, possibly by traversing the on-disk LSM tree. In the RocksdDB version with no background I/O, when a memory component is full, we discard it instead of persisting it, serving requests only from the pre-populated data store. We compare the throughput achieved by the two systems and report it in Figure 2. The plot shows that, for all workloads, background I/O represents a major performance bottleneck, yielding up to a 3x in throughput loss with respect to the ideal case.

Driven by these results, we investigate the causes that trigger frequent and intensive I/O operations. We identify three main sources of expensive I/O operations, one for each of the three main components of the LSM tree architecture, namely (1) *data-skew unawareness*, at the memory component level; (2) *premature and iterative compaction*, at the LSM tree level; (3) *duplicated writes* at the logging level.

1. Data skew unawareness. Many KV store workloads exhibit skewed data popularity, in which a few *hot* keys have a much higher probability of being updated than *cold* keys [16]. As we show in Section 5, some Nutanix production workloads also exhibit similar skew.

Data skew causes the commit log to grow more rapidly than C_m, because updates to the same keys are appended to the log but absorbed in-place by C_m. This triggers frequent flushes of C_m before it reaches its maximum size. Not only does this increase the frequency of flushing, but because the size of the flushed C_m is often smaller than the maximum, the fixed cost of opening and storing a file in L_0 is not amortized by the actual writing of data in it.

Data skew also has a negative impact on the extent of the compaction process. In fact, it is highly likely that a copy of a hot key is present in many levels of the LSM tree structure. This results in frequent compaction operations that easily trickle down the LSM tree structure, causing long cascading compaction phases at L_i that likely result into spilling new data to L_{i+1}.

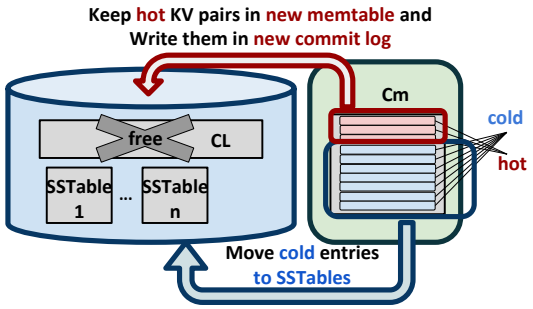


Figure 3: TRIAD-MEM: Before hot-cold key separation.

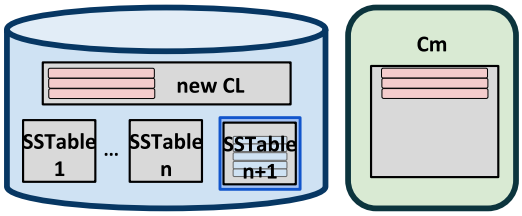


Figure 4: TRIAD-MEM: After hot-cold key separation and flush.

2. Premature and iterative compaction. Existing LSM KV systems exhibit a two-fold limitation in the compaction process. Some LSM KV stores keep only one file in L_0 to avoid looking up several SSTables in L_0 when reading [5]. As a result, every time the memory component is flushed, a compaction from L_0 to the underlying levels is triggered. This choice leads to frequent compactions of the LSM tree.

Other LSM schemes [4, 12, 14] keep several files in L_0. This approach leads to the second limitation of existing LSM KV stores. The issue lies in how LSMs compact L_0 to L_1 when several SSTables are present in L_0. In fact, files in L_0 are compacted to higher levels one at a time, resulting in several consecutive compaction operations. If two files in L_0 share a common key, this key is compacted twice in the underlying LSM tree. Data skew exacerbates this problem, because it increases the probability that multiple L_0 files contain the same set of hot keys. Clearly, the higher the load on the system, the higher is the probability of this event happening.

This phenomenon can also arise in systems that keep a single file in L_0. Indeed, during the compaction, the system continues serving user operations, thus potentially triggering multiple flushes of the memory component to L_0. As a result, when a compaction finishes, it is possible that multiple files reside in L_0.

3. Duplicate writes. When C_m is flushed to L_0, the corresponding commit log is discarded because flushing already ensures the durability of the data. Each KV pair in the new file in L_0, however, corresponds to the last version of a key written in the memory component and, hence, appended to the commit log. Therefore, when flushing the memory component to disk in L_0, the system is actually replaying I/O that it has already performed when populating the commit log.

4 TRIAD

We now provide a detailed description of TRIAD's techniques. The pseudocode for the main parts of TRIAD's algorithms are shown in Algorithm 1 and Algorithm 2. The approach we use to tackle the I/O overhead is threefold, each solution addressing one of the challenges highlighted in the previous section:

(1) **TRIAD-MEM** tackles the data skew unawareness issue at the memory component level.
(2) **TRIAD-DISK** tackles the premature and iterative compaction issue by judiciously choosing to defer and batch compaction at the disk component level.
(3) **TRIAD-LOG** tackles the duplicated writes issue, bypassing new file creation during flushes, at the commit log level.

The three techniques complement each other and target the main components of LSM KV stores. Even if they work best together, they are stand-alone techniques and generally applicable to LSM-based KV stores.

4.1 TRIAD-MEM

The goal of TRIAD-MEM is to leverage the skew exhibited by many workloads [16] to reduce flushing and, hence, reduce the frequency of compactions. To this end, TRIAD-MEM only flushes *cold* keys to disk, while keeping *hot* keys in memory. This avoids the numerous compactions triggered to ensure non-overlapping key ranges in the LSM disk structure.

TRIAD-MEM separates entries that are updated often (i.e., hot entries) from entries which are rarely updated (i.e., cold entries) upon flushing C_m to L_0. The hot entries are kept in the new C_m and only the cold entries are written to disk. This way, the hot entries are updated just in-memory and do not trigger a high number of compactions on disk. The hot-cold key separation during a flush to L_0 is shown in Figure 3 and Figure 4.

The separation between hot and cold keys is shown in the *separateKeys* function in Algorithm 2. The top-K entries of the old C_m are selected, where K is a parameter of the system. Ideally, K should be high enough to accommodate all the hot keys, but low enough to avoid a high memory overhead for C_m. Thus, properly setting K requires some *a priori* knowledge about the workload. TRIAD, however, is designed to deliver high per-

Algorithm 1 Update and Flush.

```
1: function UPDATE(Key k, Val v)
2:    Entry e =mem.getEntry(k)
3:    CommitLog log =getCommitLog()
4:    if (e≠ NULL) then
5:       e.val = v;  e.updates++
6:       CLUpdateOffset(log,&e)        ▷ Update CL name and offset in entry e
7:    else
8:       e = new Entry(k, v);  e.updates = 1
9:       CLUpdateOffset(log,&e)        ▷ Update CL name and offset in entry e
10:      mem.add(e)
11:   end if
12: end function

13: function FLUSH(Memtable mem)
14:    if (mem.getSize() < FLUSH_TH) then  ▷ Do not flush if mem too small
15:       CommitLog newLog = new CommitLog()
16:       populateLog(nwLog, mem)
17:       CommitLog log = getCommitLog()
18:       setCurrentCommitLog(newLog)
19:       discardCommigLog(log)
20:       CLUpdateOffset(newLog, mem)
21:    else
22:       Memtable hotMem = new Memtable()
23:       Memtable coldMem = new Memtable()
24:       separateKeys(mem, hotMem, coldMem)
25:       setCurrentMemtable(hotMem)
26:       CommitLog log = getCommitLog()
27:       CommitLog newLog = new CommitLog()
          ▷ Write back hotMem entries to the new log
28:       populateLog(newLog, hotMem)
29:       setCurrentCommitLog(newLog)
          ▷ Update hotMem with offsets from new CL
30:       CLUpdateOffset(newLog, hotMem)
          ▷ Extract index corresponding to cold keys
31:       CLIndex index = getKeysAndOffsets(coldMem)
          ▷ Flush only index and link it to old CL
32:       flushToDisk(index, log)
33:    end if
34: end function
```

Algorithm 2 Key Separation and Deferred Compaction.

```
1: function SEPARATEKEYS(Memtables mem, hotMem, coldMem)
2:    int hotKeyCount = sizeof(Memtable) *
                        PERC_HOT / sizeof(Entry)
3:    Entry[] hotKeys = getTopKHot(mem, hotKeyCount)
4:    hotMem.add(hotKeys)
5:    for k in hotMem do                          ▷ Reset hotness
6:       k.hotness = 0; k.updates = 0
7:    end for
8:    coldMem = mem
9:    coldMem.remove(hotKeys)
10: end function

11: function DEFERCOMPACTION(())
12:    assert(level == 0)
13:    int totalKeys = 0
14:    HyperLogLog hllVect[]
15:    FileMetaData levelFiles[]
16:    levelFiles = getLevelFiles(0)
17:    for f in levelFiles do
18:       totalKeys+= f.hllKeysCount()
19:       hllVect.pushBack(f.hllGet())
20:    end for
21:    int estimated = hhllMergedEstimate(hllVect)
22:    double overlapRatio = 1 - (estimated / totalKeys)
23:    boolean notEnoughOverlap = overlapRatio < OVERLAP_RATIO_TH
24:    boolean notEnoughFiles = getLevelFiles(0).size()≤ MAX_FILES_L0
25:    if  notEnoughOverlap ∧ notEnoughFiles  then
26:       return true                             ▷ Defer compaction
27:    end if
28:    return false
29: end function
```

formance with no information about the workload. In our current implementation, K is a constant. We will show in Section 5 that thanks to its holistic multi-level approach, TRIAD is robust against settings of K that correspond to not storing all the hot keys in memory. We are also currently investigating techniques to automatically set K depending on the runtime workload, for example by means of hill climbing [43].

The entries that are preserved in the new memory component and not sent to disk are written to the commit log associated to the new memory component, as shown in Figure 3. This write-back is necessary in order to not lose information. A final optimization when separating the hot and cold keys is not flushing at all if C_m's size is not larger than a certain threshold (denoted $FLUSH_TH$ in Algorithm 1). Indeed, in the case of very skewed workloads, a flush might be triggered not because C_m is full, but because the commit log becomes full. To avoid having a large number of small files, we keep all entries in memory, discard the old commit log, and create a new commit log, with only the freshest values of C_m entries.

We experiment with several methods for hot-cold key detection, including looking at mean and standard deviation of the update frequencies, and selection according to quantiles. Simply selecting as hot keys those keys that are updated with higher frequency than the average one is effective in all workloads.

4.2 TRIAD-DISK

TRIAD-DISK acts at L_0 of the LSM disk component. In a nutshell, TRIAD-DISK delays compaction until there is enough key overlap in the files that are being compacted. To approximate the key overlap between files, we use the HyperLogLog (HLL) probabilistic cardinality estimator [28, 30]. HLL is an algorithm that approximates the number of distinct elements in a multiset. To compute the overlap between a set of files, we define a metric we call the *overlap ratio*. Assuming we have n files on L_0, the overlap ratio is defined as 1 - ($UniqueKeys(file_1, file_2, \ldots file_n)$) / $sum(Keys(file_i))$, where $Keys(file_i)$ is the number of keys of the i-th SSTable and $UniqueKeys$ is the number of unique keys after merging the n files. $UniqueKeys$ and $Key(file_i)$ are approximated using HLL.

Figure 5 shows an example of how the overlap ratio is used to defer compaction. In the upper part of the figure, there is only one file on L_0; the L_0 file overlaps with two files on L_1. Since the overlap ratio is smaller than the cutoff threshold in this case, compaction is deferred. The lower part of the figure shows the system at a time when L_0 contains two files. The overlap ratio is computed between *all the files* in L_0 and their respective overlapping files on L_1. The overlap ratio is higher than the threshold,

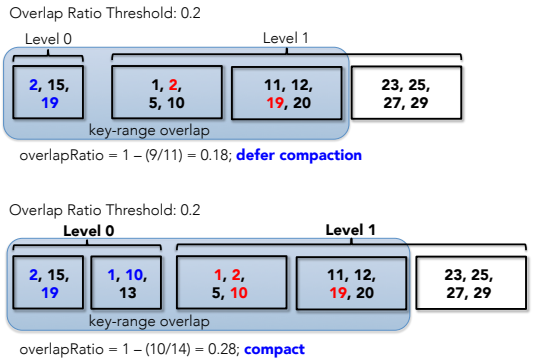


Figure 5: Overlap ratio example.

so compaction can proceed, by doing a multi-way merge between all files in L_0 and the overlapping files in L_1.

The function *deferCompaction* in Algorithm 2 shows the TRIAD-DISK pseudo-code. We associate an HLL structure to each L_0 file in the LSM disk component. Before each compaction in L_0, we calculate the overlap ratio of all files in L_0. If the overlap ratio is below a threshold, we defer the compaction, unless the number of files in L_0 exceeds the maximum allowed number. If the maximum number of files in L_0 is reached, we proceed with the L_0 to L_1 compaction, regardless of the key overlap.

The use of HLL is not new in the context of LSM compaction. So far, however, the way HLL is used is to detect which files have the most key overlap to be compacted (for instance in systems such as Cassandra [1]). This way, the highest number of duplicate keys is discarded during compaction. RocksDB employs a similar idea, where the estimation of the key overlap in files at L_i and L_{i+1} is based on the files' key ranges and sizes. Our use of HLL is different. Instead of employing HLL to decide which files to compact, we are using HLL to decide *whether* to compact L_0 into L_1 at the current moment, or defer it to a later point in time. If the L_0 and L_1 SSTables do not have enough key overlap, compaction is delayed until more L_0 SSTables are generated.

Current LSMs trigger the compaction of L_0 into L_1 as soon as the number of files on L_0 reaches a certain threshold. The larger the threshold, the more files need to be accessed in L_0 by read operations, which increases read latency. However, since the chance of a key being present multiple files on L_0 is low (otherwise, the large overlap ratio would trigger compaction), TRIAD-DISK can tolerate more files in L_0 without hurting read performance, as we show in Section 5.

4.3 TRIAD-LOG

The main insight of TRIAD-LOG is that the data that is written to memory and then persisted into L_0 SSTables is already written to disk, in the commit log. The general idea of TRIAD-LOG is to transform CL into a special type of L_0 SSTable, a *CL-SSTable*. This way, flushing from memory to L_0 is avoided altogether.

TRIAD-LOG enhances the role played by the commit log. As C_m is being written to, the commit log plays its classic role. When flushing is triggered, instead of copying C_m to disk, we convert the commit log into a CL-SSTable. As shown in Figure 6, instead of storing copies of the memory components in L_0, we store CL-SSTables. For readability, we only depict the TRIAD-LOG technique, and not the integration with our two other techniques.

The advantage of treating the commit logs as L_0 SSTables is that the I/O due to flushing from memory is avoided. However, unlike SSTables, the commit log is not sorted. The sorted structure of the classic SSTables makes it easy to merge SSTables during compaction and to retrieve information from the files. To avoid scanning the entire CL-SSTable in order to find an entry in L_0 files, we keep the commit log file offset of the most recent update in C_m, for each KV pair. Once the flush operation is triggered, only the small index associated to the offsets in the commit log is written to disk. The index is then grouped with its corresponding commit log file, thus creating the new L_0 CL-SSTable format.

For instance, consider a commit log with entries of size 8B, in the format (Key; Value): (1;10), (2;20), (3;30), (4;40), (3;300). Then, in C_m, TRIAD-LOG keeps the following entries, in the format (Key; Value; CL offset; CL name): (1; 10; 0; CL-name), (2; 20; 8; CL-name), (3; 300; 32; CL-name), and (4; 40; 24; CL-name). The CL offset is equal to 32 for Key 3, because we keep the offset of the most recent update.

TRIAD-LOG offers the greatest benefits when the workload is more uniform. For such workloads it is relatively rare that the same key appears several times in the log. The corresponding CL-SSTable therefore contains the most recent values of many distinct keys, and relatively few older values. For skewed workloads, in contrast, the log typically contains multiple updates of the same keys, and the corresponding CL-SSTable therefore stores a high number of old values that are no longer relevant.

The flow of the write operation remains unchanged by TRIAD-LOG. The writes are performed in C_m and persisted in the commit log. The only difference is that apart from the value associated to the key, the commit log name and offset entries are updated as well. Similarly, the read path is largely unchanged, except for accessing the files in L_0. As before, the reads first look in C_m, then in all of the L_0 files, and then in one file for each of the lower levels of the disk component. Unlike before, when a file from L_0 is read, the index is searched

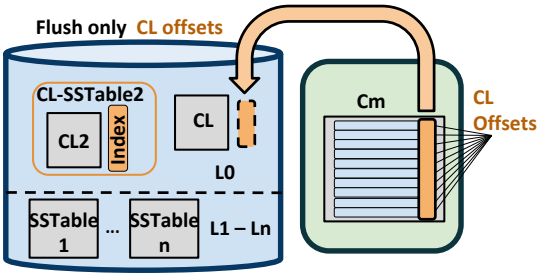


Figure 6: TRIAD-LOG operation flow.

for the key, and, if found, the CL-SSTable is accessed at the corresponding offset.

Compaction from L_1 to L_n is unchanged, because no modifications are done to the SSTable format on these levels. Only the compaction between L_0 and L_1 is affected by our technique. A new compaction operation is needed for merging a CL-SSTable with a regular SSTable. Since the index kept on the CL-SSTable is sorted, it is still possible to proceed in a merge-sort style manner. For clarity and brevity of the presentation, we omit the pseudocode for merging SSTables.

It is straightforward to integrate TRIAD-LOG with TRIAD-DISK, since TRIAD-DISK affects only the decision to call compaction. The integration with TRIAD-MEM is done by flushing only the part of the index corresponding to the cold keys, ignoring the offsets of the hot keys. Then, during compaction, the hot keys are skipped, similarly to the duplicate updates.

TRIAD Memory Overhead Analysis. TRIAD reduces I/O by using additional metadata in memory. TRIAD-MEM needs an *update frequency* (4B) field for each memory component entry. For each (CL-)SSTable on L_0, TRIAD-DISK tracks the HLL structure (4KB per file). TRIAD-LOG adds two new fields for each memory component entry: the *commit log file ID* (4B) and the *offset* in the commit log (4B). Finally, TRIAD-LOG keeps track of the offsets index (8B per entry) for each CL-SSTable on L_0. While the HLLs and offset indexes could be stored on disk, this would incur a performance penalty. Since the number of files on L_0 is not large, the memory overhead is not significant. Generally, less than 10 files are kept on L_0. Hence, an upper bound on TRIAD's memory overhead is: $12B * EntriesC_m + 10 * (4KB + EntriesC_m * 8B)$. In our tests, the memory overhead is on the order of tens of MB, which is negligible with respect to the tens of GB of I/O saved.

5 Evaluation

We implement TRIAD as an extension of Facebook's popular RocksDB LSM-based KV store. The source code of our implementation is available at `https://github.com/epfl-labos/TRIAD`. We evaluate TRIAD with production and synthetic workloads, and we compare it against RocksDB. We show that:

(1) TRIAD achieves up to 193% higher throughput in production workloads.
(2) TRIAD effectively reduces I/O by an order of magnitude and spends, on average, 77% less time performing flushing and compaction.
(3) TRIAD's three techniques work in synergy and enable the system to achieve high throughput without a priori information about the workload (e.g., skew on data popularity or write intensity).

5.1 Experimental Setup

We compare TRIAD against RocksDB. Unless stated otherwise, RocksDB is configured to run with its default parameters, and we do not change the corresponding values in the TRIAD implementation. TRIAD uses an overlap threshold of 0.4 and a maximum number of 6 L_0 files for TRIAD-DISK. In addition, we configure TRIAD-MEM such that its definition of hot keys corresponds to the top 1 percent of keys in terms of access frequency.

To evaluate TRIAD, we use four production workloads from Nutanix (see Section 5.2). We complement our evaluation with synthetic benchmarks that allow us to control key parameters of the workload, such as skew and write intensity (see Section 5.3). The evaluation is performed on a 20 core Intel Xeon, with two 10-core 2.8 GHz processors, 256 GB of RAM, 960GB SSD Samsung 843T, running Ubuntu 14.04.

Each synthetic benchmark experiment consists of a number of threads concurrently performing operations on the KV store – searching, inserting or deleting keys. Each operation is chosen at random, according to the given workload probability distribution, and performed on a key chosen according to the given workload skew distribution. Before each experiment, the LSM tree is initialized with roughly half of the keys in the key range.

We use as evaluation metrics throughput measured in KOperations/second (KOPS), bytes written to disk, time spent in background operations (i.e., compaction and flushing), write amplification (WA), and read amplification (RA). WA and RA are established metrics for measuring I/O LSM KV store efficiency [4, 12, 32, 34, 35, 38, 48]. WA is the amount of data written to storage compared to the amount of data that the application writes. Intuitively, the lower the WA, the less work is done during compaction. We compute system-wide WA as: $WA = (Bytes_{flushed} + Bytes_{compacted})/Bytes_{flushed}$. RA is the average number of disk accesses per read.

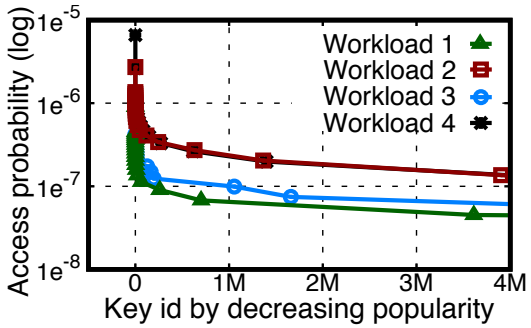


Figure 7: Production workloads key probability distributions (Logarithmic scale on y axis).

	Wkld. 1	Wkld. 2	Wkld. 3	Wkld. 4
Updates	250M	75M	200M	75M
Keys	40M	9M	30M	8M

Figure 8: Production workloads: number of updates and number of keys.

5.2 Production Workloads

The production workloads used for the evaluation of TRIAD are internal Nutanix metadata workloads. The key probability distributions of the workloads are shown in Figure 7. The data sizes and number of updates are shown in Figure 8. The production workloads have two different skew profiles: W2 and W4 have more skew in their access patterns, W1 and W3 have less skew.

The left-hand side of Figure 9A presents the throughput comparison between RocksDB and TRIAD, for the four production workloads. TRIAD outperforms RocksDB in the four workloads, with a throughput increase of up to 193%. The right-hand side of Figure 9A shows the corresponding WA for each of the workloads. TRIAD reduces WA by up to 4x.

As expected [34], in RocksDB the WA is higher for the less skewed workloads (W1 and W3) and lower for the more skewed workloads (W2 and W4). There is also a clear correlation between the throughput and the WA: throughput is lower in the workloads with higher WA.

For TRIAD WA is uniform across the four workloads, because TRIAD-MEM converts the skew of the application workload into a disk workload that is closer to uniform. Hence, the workload skew perceived by the disk component is more or less the same across the four workloads, leading, in turn, to more predictable throughput. In contrast with RocksDB, TRIAD's throughput does not exhibit high fluctuation across workloads. We explore this connection between throughput and WA further in the next section.

5.3 Synthetic Workloads

We define three workload skew profiles: (WS1) A highly skewed workload where 1% of the data is accessed and updated 99% of the time. This workload reflects the characteristics of Facebook workloads analyzed in [16]. (WS2) A medium skew workload, where 20% of the data is accessed and updated 80% of the time. (WS3) A uniform workload where all keys have the same popularity.

We use two different read-write ratios: one with 10% reads and 90% writes, and one with 50% reads and 50% writes. In all experiments, each key is 8B and each value is 255B. To shorten our experiments with the synthetic workloads, we use a small memory component of 4MB and a dataset of 1M keys, so that compactions happen at shorter time intervals.

Figure 9B shows the throughput comparison between TRIAD and RocksDB for the three workload skews and two read-write ratios. Figure 9C shows the corresponding WA. TRIAD performs up to 2.5x better than RocksDB for the skewed workloads and up to 2.2x better for the uniform workloads.

For WS1 all the hot data fits in memory, allowing TRIAD to achieve a throughput increase of 50% for both the write-intensive and the balanced workloads. For WS2, TRIAD-MEM cannot accommodate all the hot keys. Nevertheless, TRIAD still achieves a throughput gain of 51% in the write-intensive workload and 25% in the balanced workload, because TRIAD-DISK and TRIAD-LOG act as a safety net against possible undersizing of the data structure tracking hot keys (Section 4.1), due to lack of detailed knowledge of the workload characteristics. This result showcases the robustness of TRIAD: It consistently delivers high performance, despite not having any prior knowledge of the incoming workload.

WA is decreased by up to 4x in the moderately skewed and uniform workloads. For the highly skewed workload, however, the WA does not change, despite the gain in throughput. This happens because the 1% of the data that is updated 99% of the time fits entirely in memory. As a consequence, C_m is only rarely flushed (as we explain in Section 4), because it takes longer for enough cold entries to be present in C_m to trigger a flush. Therefore, even if the total number of bytes is decreased by an order of magnitude, as Figure 9D shows on the left-hand side, the proportion between the compacted bytes and flushed bytes is similar to RocksDB.

Finally, the right-hand side of Figure 9D shows the time spent in compaction. For the highly skewed workload, the time spent in compaction in TRIAD is an order of magnitude lower than RocksDB, for the same reason as explained above. For the moderately skewed and

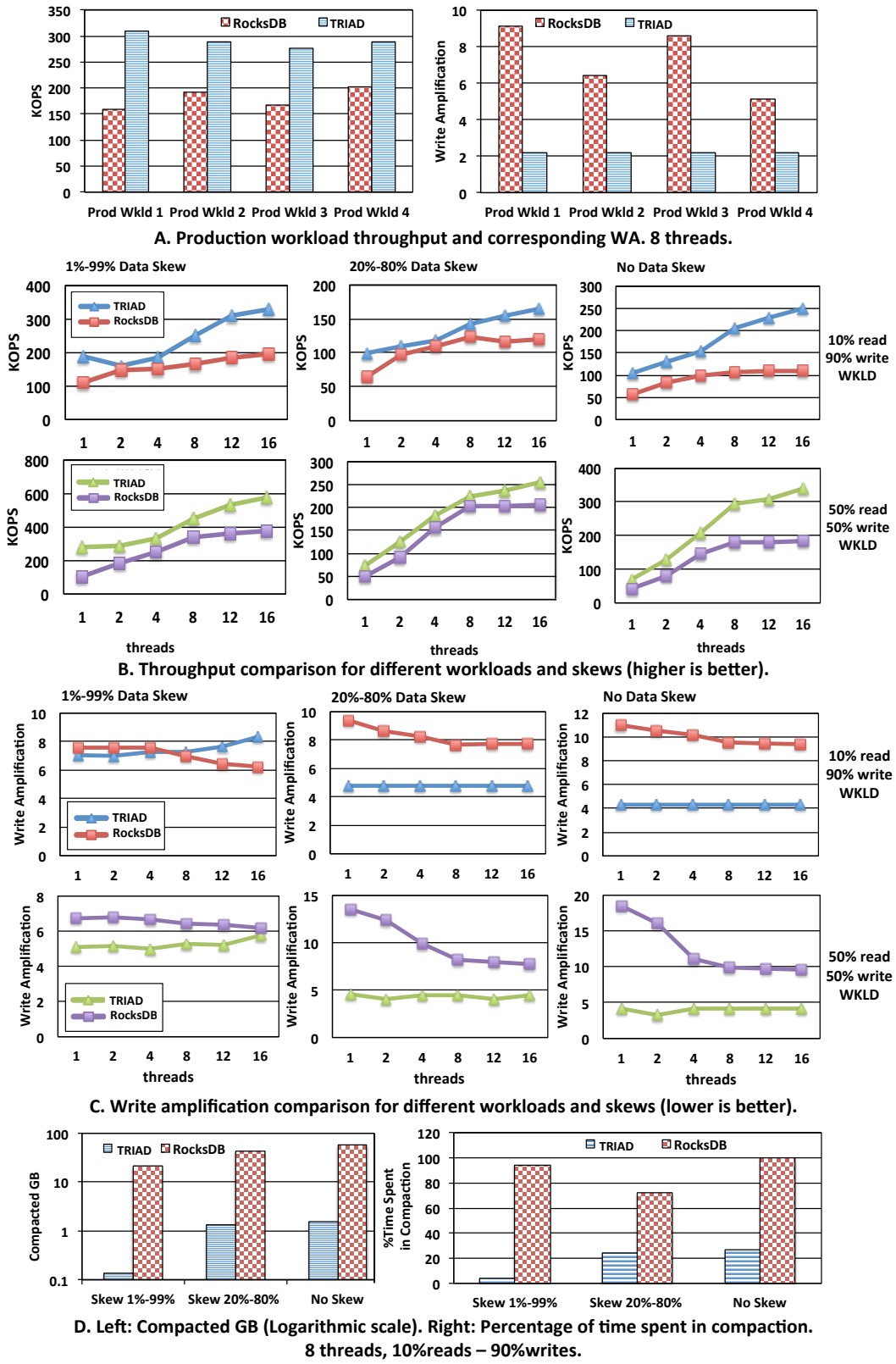


Figure 9: TRIAD in production and synthetic workloads.

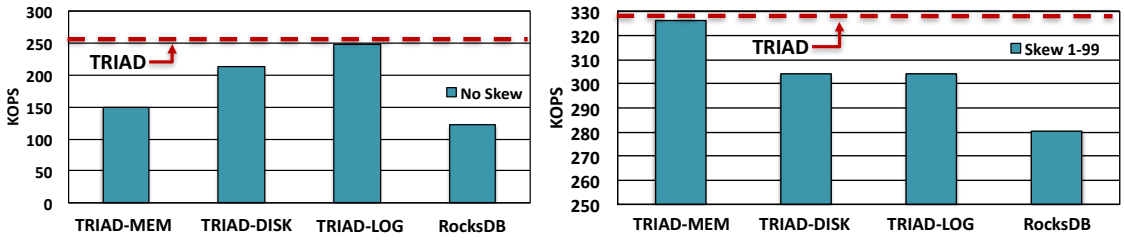


Figure 10: Throughput breakdown for uniform and skewed workloads. 16 threads.

uniform workloads, the time spent in compaction is decreased by 48% and 72%, respectively.

5.4 Breakdown of TRIAD's Benefits

We discuss the contribution of each of TRIAD's techniques, for different types of workloads, reporting the throughput achieved by versions of TRIAD where we only implement one out of the three techniques.

Figure 10 shows the throughput breakdown for each of the techniques, for synthetic workloads WS3 (left-hand side) and WS1 (right-hand side), with a 10%–90% read–write ratio. While all three techniques outperform RocksDB individually, TRIAD-MEM brings more benefits than TRIAD-DISK and TRIAD-LOG for the highly skewed workload, and vice-versa for the uniform workload. Indeed, TRIAD-MEM alone obtains 97% of the throughput that TRIAD achieves for the skewed workload. For the uniform workload, TRIAD-DISK and TRIAD-LOG obtain 85% and 97%, respectively.

A similar trend can be noticed in the WA breakdown in Figure 11. TRIAD-MEM performs best for WS1, but does not decrease WA as the workload is closer to uniform, having close to no effect compared to RocksDB for the workload with no skew (right-most column). TRIAD-DISK and TRIAD-LOG are complementary to TRIAD-MEM, decreasing WA by up to 60% and 40%, respectively, for the uniform workload.

The lower-right plot in Figure 11 shows the RA breakdown for a uniform workload, with 10% reads. As expected, TRIAD-MEM lowers RA, because more requests can be served from memory. TRIAD-DISK, however, increases RA compared to the baseline, as it keeps more files in L_0, and all these files may have to be accessed on a read. TRIAD-LOG does not have an impact on read amplification. Overall, TRIAD has a low overhead over the baseline, increasing RA by at most 5%.

The breakdown shows tat the three techniques are complementary: no one alone gives 100% of the benefits across all workload types. Their combination allows TRIAD to achieve high performance for any workload, automatically adapting to its characteristics without a priori knowledge.

6 Related Work

Our work is related to previous designs of LSM-based KV stores and to various systems that employ optimization techniques similar to the ones integrated in TRIAD.

Related LSM-based KV stores. LevelDB [4] is one of the earliest LSM-based KV stores and employs level-style compaction. Its single-threaded compaction, along with the use of a global lock for synchronization at the memory component level are two of its main bottlenecks. RocksDB [12] introduces multi-threaded compaction and tackles other concurrency issues. LevelDB and RocksDB expose several tuning knobs, such as the number and the sizing of levels, and policies for compaction [13, 14, 27]. Recent studies, simulations and analytical models show that the efficiency of LSM-based KV stores is highly dependent on their proper setting, as well as workload parameters [34, 26] and requirements like memory budget [24]. In contrast, TRIAD presents techniques that cover a large spectrum of workloads. Thanks to its holistic approach, TRIAD is able to deliver high performance without relying on *a priori* information about the workload.

bLSM [44] proposes carefully scheduling compaction to bound write latency. VT-tree [45] uses an extra layer of indirection to avoid sorting any previously sorted KV pairs during compaction. HyperlevelDB [9] also addresses the write and compaction issues in LevelDB, through improved parallelism and an alternative compaction algorithm [7, 8]. HyperLevelDB's compaction chooses a set of SSTables which result in the lowest WA between two levels. TRIAD takes a different approach to prevent the occurrence of high WA, by using HLL to decide whether to compact or not at the first level of the disk component.

LSM-trie [47] proposes a compaction scheme based on the use of cryptographic functions. This scheme gives up the sorted order of the entries in the LSM tree to favor compaction efficiency over performance in range queries. TRIAD instead preserves the sorted order of the keys, facilitating support for efficient range queries.

WiscKey [37] separates keys from values and only stores keys in a sorted LSM tree, allowing it to reduce

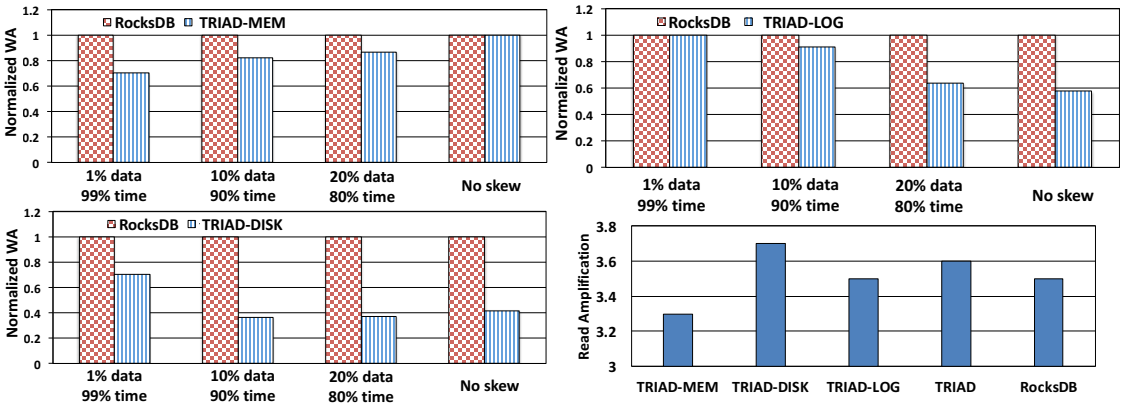


Figure 11: Write Amplification and Read Amplification Breakdown. 8 threads.

data movement, and consequently reduce write amplification. The techniques proposed in TRIAD are orthogonal to this approach, and can be leveraged in synergy to it to further enhance I/O efficiency.

Cassandra [1], HBase [11] and BigTable [22] are distributed LSM KV stores, employing size-tiered compaction. In addition, Cassandra also supports the leveled compaction strategy, based on LevelDB's compaction scheme [5, 6]. For both compaction strategies, Cassandra introduced HyperLogLog (HLL) to estimate the overlap between SSTables, before starting the merge [10]. TRIAD also makes use of HLL in its deferred compaction scheme. However, instead of using HLL to determine which files to compact, the overlap between files computed with HLL is used only at L_0, to determine whether we should compact or wait.

Ahmad and Kemme [15] also target a distributed KV store and propose to offload the compaction phase to dedicated servers. In contrast, the techniques proposed in TRIAD are applied within each single KV store instance and do not need dedicated resources to be implemented.

Tucana [42], LOCS [46] and FloDB [17] act on other aspects of the KV store design to improve performance. Tucana uses an internal structure similar to a $B-\varepsilon$ tree [20] and uses copy-on-write instead of write-ahead logging. LOCS exploits the knowledge of the underlying SSD multi-channel architecture to improve performance, e.g., by load balancing I/O. FloDB inserts an additional fast in-memory buffer on top of the existing in-memory component of the LSM tree to achieve better scalability. TRIAD can integrate some of these features to further improve its performance.

Systems with similar optimization techniques. The hot-cold separation technique is employed in SSDs to improve the efficiency of the garbage collection needed by the Flash Translation Layer [23, 33]. In TRIAD, instead, it is used at the KV store level to reduce the amount of data written to disk.

Delaying and batching the execution of updates is used in $B-\varepsilon$ trees and in systems, e.g., file-systems [31], which use $B-\varepsilon$ trees as main building block. This technique is employed to amortize the cost of updates [19] and to reduce the cases in which the effect of an update is immediately undone by a following update [49]. By contrast, TRIAD defers the compaction of the L_0 level of the LSM tree and batches the compaction of multiple keys to increase the efficiency of the compaction process.

7 Conclusion

TRIAD is a new LSM KV store aiming to reduce background I/O operations to disk. TRIAD embraces a holistic approach that operates at different levels of the LSM KV store architecture. TRIAD increases I/O efficiency by incorporating data skew awareness, by improving the compaction process of the LSM tree data structure and by performing more efficient logging.

We compared TRIAD with Facebook's RocksDB and we showed, using production and synthetic workloads, that TRIAD achieves up to an order of magnitude lower I/O overhead and up to 193% higher throughput.

Acknowledgements. We would like to thank our shepherd, Liuba Shrira, the anonymous reviewers and Dmitri Bronnikov, Rishi Bhardwaj, Ashvin Goel, and Amitabha Roy for their feedback that helped us to improve the paper. This work was supported in part by the Swiss National Science Foundation through grant No. 166306 and by a gift from Nutanix, Inc. Part of the work has been done while Oana Balmau was an intern at Nutanix.

References

[1] Apache Cassandra. `http://cassandra.apache.org`.

[2] Memcached, an open source, high-performance, distributed memory object caching system. `https://memcached.org/`.

[3] Redis, an open source, in-memory data structure store. https://redis.io/.

[4] LevelDB, a fast and lightweight key/value database library by Google, 2005. https://github.com/google/leveldb.

[5] Leveled Compaction in Apache Cassandra, 2011. http://www.datastax.com/dev/blog/leveled-compaction-in-apache-cassandra.

[6] When to Use Leveled Compaction, 2012. http://www.datastax.com/dev/blog/when-to-use-leveled-compaction.

[7] Hyperleveldb performance benchmarks., 2013. http://hyperdex.org/performance/leveldb/.

[8] Inside hyperleveldb., 2013. http://hackingdistributed.com/2013/06/17/hyperleveldb/.

[9] HyperLevelDB, a fork of LevelDB intended to meet the needs of HyperDex while remaining compatible with LevelDB., 2014. https://github.com/rescrv/HyperLevelDB.

[10] Improving compaction in Cassandra with cardinality estimation, 2014. http://www.datastax.com/dev/blog/improving-compaction-in-cassandra-with-cardinality-estimation.

[11] Apache HBase, a distributed, scalable, big data store, 2016. http://hbase.apache.org/.

[12] RocksDB, a persistent key-value store for fast storage environments, 2016. http://rocksdb.org/.

[13] RocksDB options of compaction priority, 2016. http://rocksdb.org/blog/2016/01/29/compaction_pri.html.

[14] RocksDB tuning guide, 2016. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.

[15] AHMAD, M. Y., AND KEMME, B. Compaction management in distributed key-value datastores. *Proc. VLDB Endow. 8*, 8 (Apr. 2015), 850–861.

[16] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review* (2012), vol. 40.

[17] 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* (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 80–94.

[18] BĂSESCU, C., CACHIN, C., EYAL, I., HAAS, R., SORNIOTTI, A., VUKOLIĆ, M., AND ZACHEVSKY, I. Robust data sharing with key-value stores. DSN 2012.

[19] BENDER, M. A., FARACH-COLTON, M., JANNEN, W., JOHNSON, R., KUSZMAUL, C., B., PORTER, D. E., YUAN, J., AND ZHAN, Y. An introduction to bε-trees and write-optimization. *login 40*, 5 (Oct. 2015).

[20] BRODAL, G. S., AND FAGERBERG, R. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, 2003), SODA '03, Society for Industrial and Applied Mathematics, pp. 546–554.

[21] CANO, I., AIYAR, S., AND KRISHNAMURTHY, A. Characterizing private clouds: A large-scale empirical analysis of enterprise clusters. SOCC 2016.

[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. *ACM Transactions on Computer Systems 26*, 2 (2008).

[23] CHANG, L.-P. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the 2007 ACM Symposium on Applied Computing* (New York, NY, USA, 2007), SAC '07, ACM, pp. 1126–1130.

[24] DAYAN, N., ATHANASSOULIS, M., AND IDREOS, S. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data* (New York, NY, USA, 2017), SIGMOD '17, ACM, pp. 79–94.

[25] 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. SOSP 2007.

[26] DONG, S., CALLAGHAN, M., GALANIS, L., BORTHAKUR, D., SAVOR, T., AND STRUM, M. Optimizing space amplification in rocksdb. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings* (2017).

[27] DONG, S., CALLAGHAN, M., GALANIS, L., BORTHAKUR, D., SAVOR, T., AND STUMM, M. Optimizing space amplification in rocksdb.

[28] FUSY, É., OLIVIER, G., AND MEUNIER, F. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. AofA 2007.

[29] GOLAN-GUETA, G., BORTNIKOV, E., HILLEL, E., AND KEIDAR, I. Scaling concurrent log-structured data stores. Eurosys 2015.

[30] HEULE, S., NUNKESSER, M., AND HALL, A. Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. ICDT 2013.

[31] JANNEN, W., YUAN, J., ZHAN, Y., AKSHINTALA, A., ESMET, J., JIAO, Y., MITTAL, A., PANDEY, P., REDDY, P., WALSH, L., BENDER, M. A., FARACH-COLTON, M., JOHNSON, R., KUSZMAUL, B. C., AND PORTER, D. E. Betrfs: Write-optimization in a kernel file system. *Trans. Storage 11*, 4 (Nov. 2015), 18:1–18:29.

[32] KUSZMAUL, B. A comparison of fractal trees to log-structured merge (lsm) trees. *White Paper* (2014).

[33] LEE, S., LIU, M., JUN, S., XU, S., KIM, J., AND ARVIND, A. Application-managed flash. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies* (Berkeley, CA, USA, 2016), FAST'16, USENIX Association, pp. 339–353.

[34] LIM, H., ANDERSEN, D. G., AND KAMINSKY, M. Towards accurate and fast evaluation of multi-stage log-structured designs. FAST 2016.

[35] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. Silt: A memory-efficient, high-performance key-value store. SOSP 2011.

[36] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. MICA: A holistic approach to fast in-memory key-value storage. *management 15*, 32 (2014).

[37] LU, L., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Wisckey: Separating keys from values in ssd-conscious storage. FAST 2016.

[38] MARMOL, L., SUNDARARAMAN, S., TALAGALA, N., AND RANGASWAMI, R. Nvmkv: A scalable, lightweight, ftl-aware key-value store. USENIX ATC 2015.

[39] MARMOL, L., SUNDARARAMAN, S., TALAGALA, N., RANGASWAMI, R., DEVENDRAPPA, S., RAMSUNDAR, B., AND GANESAN, S. Nvmkv: A scalable and lightweight flash aware key-value store. HotStorage 2014.

[40] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The log-structured merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996).

[41] OUSTERHOUT, J., AND DOUGLIS, F. Beating the I/O bottleneck: A case for log-structured file systems. *ACM SIGOPS Operating Systems Review 23*, 1 (1989).

[42] PAPAGIANNIS, A., SALOUSTROS, G., GONZÁLEZ-FÉREZ, P., AND BILAS, A. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference* (Berkeley, CA, USA, 2016), USENIX ATC '16, USENIX Association, pp. 537–550.

[43] RUSSELL, S., AND NORVIG, P. Artificial intelligence: a modern approach (2nd edition).

[44] SEARS, R., AND RAMAKRISHNAN, R. bLSM: a general purpose log structured merge tree. SIGMOD/PODS 2012, ACM.

[45] SHETTY, P., SPILLANE, R., MALPANI, R., ANDREWS, B., SEYSTER, J., AND ZADOK, E. Building workload-independent storage with vt-trees. FAST 2013.

[46] 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* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 16:1–16:14.

[47] WU, X., XU, Y., SHAO, Z., AND JIANG, S. LSM-trie: An LSM-tree-based ultra-large key-value store for small data items. USENIX ATC 2015.

[48] YANG, J., PLASSON, N., GILLIS, G., TALAGALA, N., AND SUNDARARAMAN, S. Don't stack your log on my log. INFLOW 2014.

[49] YUAN, J., ZHAN, Y., JANNEN, W., PANDEY, P., AKSHINTALA, A., CHANDNANI, K., DEO, P., KASHEFF, Z., WALSH, L., BENDER, M. A., FARACH-COLTON, M., JOHNSON, R., KUSZMAUL, B. C., AND PORTER, D. E. Optimizing every operation in a write-optimized file system. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies* (Berkeley, CA, USA, 2016), FAST'16, USENIX Association, pp. 1–14.

Engineering Record And Replay For Deployability

Robert O'Callahan* Chris Jones* Nathan Froyd Kyle Huey*
Mozilla Corporation
Albert Noll* Nimrod Partush*
Swisscom AG *Technion*

Abstract

The ability to record and replay program executions with low overhead enables many applications, such as reverse-execution debugging, debugging of hard-to-reproduce test failures, and "black box" forensic analysis of failures in deployed systems. Existing record-and-replay approaches limit deployability by recording an entire virtual machine (heavyweight), modifying the OS kernel (adding deployment and maintenance costs), requiring pervasive code instrumentation (imposing significant performance and complexity overhead), or modifying compilers and runtime systems (limiting generality). We investigated whether it is possible to build a practical record-and-replay system avoiding all these issues. The answer turns out to be yes — if the CPU and operating system meet certain non-obvious constraints. Fortunately modern Intel CPUs, Linux kernels and user-space frameworks do meet these constraints, although this has only become true recently. With some novel optimizations, our system RR records and replays real-world low-parallelism workloads with low overhead, with an entirely user-space implementation, using stock hardware, compilers, runtimes and operating systems. RR forms the basis of an open-source reverse-execution debugger seeing significant use in practice. We present the design and implementation of RR, describe its performance on a variety of workloads, and identify constraints on hardware and operating system design required to support our approach.

1 Introduction

The ability to record a program execution with low overhead and play it back precisely has many applications [14, 15, 19] and has received significant attention in the research community. It has even been implemented in products such as VMware Workstation [28], Simics [20], UndoDB [1] and TotalView [22]. Unfortunately, deployment of these techniques has been limited, for various reasons. Some approaches [17, 20, 28] require recording and replaying an entire virtual machine, which is heavyweight. Other approaches [6, 14, 26, 29] require running a modified OS kernel, hindering deployment and adding security and stability risk to the system. Requiring compiler and language runtime changes [29] also hinders deployment, especially when applications include their own JIT compilers. Some approaches [24, 30, 35] require custom hardware not yet available. Many approaches [1, 7, 22, 34] require pervasive instrumentation of code, which adds complexity and overhead, especially for self-modifying code (commonly used in polymorphic inline caching [25] and other implementation techniques in modern just-in-time compilers). A performant dynamic code instrumentation engine is also expensive to build and maintain.

We set out to build a system that maximizes deployability by avoiding all these issues: to record and replay unmodified user-space applications with stock Linux kernels, compilers, language runtimes, and x86/x86-64 CPUs, with a fully user-space implementation running without special privileges, and without using pervasive code instrumentation. We assume RR should run unmodified applications, and they will have bugs (including data races) that we wish to faithfully record and replay, but these applications will not maliciously try to subvert recording or replay. We combine techniques already known, but not previously demonstrated working together in a practical system: primarily, using `ptrace` to record and replay system call results and signals, avoiding non-deterministic data races by running only one thread at a time, and using CPU hardware performance counters to measure application progress so asynchronous signal and context-switch events are delivered at the right moment [33]. Section 2 describes our approach in more detail.

With that in place, we discovered the main perfor-

*Majority of work carried out while supported by Mozilla Research.

mance bottleneck for low-parallelism workloads was context switching induced by using `ptrace` to monitor system calls. We implemented a novel *in-process system-call interception* technique to eliminate those context switches, dramatically reducing recording and replay overhead on important real-world workloads. This optimization relies on modern Linux kernel features: `seccomp-bpf` to selectively suppress `ptrace` traps for certain system calls, and `perf` context-switch events to detect recorded threads blocking in the kernel. Section 3 describes this work, and Section 4 gives some performance results, showing that on important application workloads RR recording and replay slowdown is less than a factor of two.

We rely on hardware and OS features designed for other goals, so it is surprising that RR works. In fact, it skirts the edge of feasibility; in particular it cannot be implemented on ARM CPUs. Section 5 summarizes RR's hardware and software requirements, which we hope will influence system designers.

RR is in daily use by many developers as the foundation of an efficient reverse-execution debugger that works on complex applications such as Samba, Firefox, Chromium, QEMU, LibreOffice and Wine. It is free software, available at `https://github.com/mozilla/rr`. This paper makes the following research contributions:

- We show that record and replay of user-space processes on modern, stock hardware and software without pervasive code instrumentation is possible and practical.
- We introduce an *in-process system-call interception* technique and show it dramatically reduces overhead.
- We show that for low-parallelism workloads, RR recording and replay overhead is reasonably low, lower than other approaches with comparable deployability.
- We identify hardware and operating system design constraints required to support our approach.

An extended technical report containing additional technical details and a retrospective on "lessons learned" during the development and use of RR is available [32].

2 Design

2.1 Summary

Most low-overhead record-and-replay systems depend on the observation that CPUs are mostly deterministic. We identify a boundary around state and computation, record all sources of nondeterminism within the boundary and all inputs crossing into the boundary, and reexecute the computation within the boundary by replaying the nondeterminism and inputs. If all inputs and nondeterminism have truly been captured, the state and computation within the boundary during replay will match that during recording.

To enable record and replay of arbitrary Linux applications, without requiring kernel modifications or a virtual machine, RR records and replays the user-space execution of a group of processes. To simplify invariants, and to make replay as faithful as possible, replay preserves almost every detail of user-space execution. In particular, user-space memory and register values are preserved exactly, with a few exceptions noted later in the paper. This implies CPU-level control flow is identical between recording and replay, as is memory layout.

While replay preserves user-space state and execution, only a minimal amount of kernel state is reproduced during replay. For example, file descriptors are not opened, signal handlers are not installed, and filesystem operations are not performed. Instead the recorded user-space-visible effects of those operations, and future related operations, are replayed. We do create one replay thread per recorded thread (not strictly necessary), and we create one replay address space (i.e. process) per recorded address space, along with matching memory mappings.

With this design, our recording boundary is the interface between user-space and the kernel. The inputs and sources of nondeterminism are mainly the results of system calls, and the timing of asynchronous events.

2.2 Avoiding Data Races

With threads running on multiple cores, racing read-write or write-write accesses to the same memory location by different threads would be a source of nondeterminism. Therefore we take the common approach [17, 28, 1, 15] running only one thread at a time. RR preemptively schedules these threads, so context switch timing is nondeterminism that must be recorded. Data race bugs can still be observed if a context switch occurs at the right point in the execution (though bugs due to weak memory models cannot be observed).

This approach is much simpler and more deployable than alternatives [7, 18, 34, 39, 29], avoids assuming programs are race-free [14, 29], and is efficient for low-parallelism workloads. There is a large slowdown for workloads with a consistently high degree of parallelism; however, even for applications which are potentially highly parallel, users often apply RR to test workloads with relatively small datasets and hence limited parallelism.

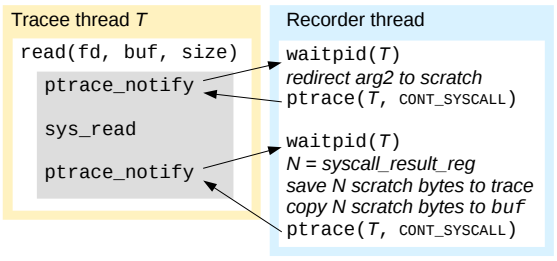


Figure 1: Recording simple system call

2.3 System Calls

System calls return data to user-space by modifying registers and memory, and these changes must be recorded. The `ptrace` system call allows a process to supervise the execution of other "tracee" processes and threads, and to be synchronously notified when a tracee thread enters or exits a system call. When a tracee thread enters the kernel for a system call, it is suspended and RR is notified. When RR chooses to run that thread again, the system call will complete, notifying RR again, giving it a chance to record the system call results. RR contains a model of most Linux system calls describing the user-space memory they can modify, given the system call input parameters and result.

As noted above, RR normally avoids races by scheduling only one thread at a time. However, if a system call blocks in the kernel, RR must try to schedule another application thread to run while the blocking system call completes. It's possible (albeit unlikely) that the running thread could access the system call's output buffer and race with the kernel's writes to that buffer. To avoid this, we redirect system call output buffers to per-thread temporary "scratch memory" which is otherwise unused by the application. When we get a `ptrace` event for a blocked system call completing, RR copies scratch buffer contents to the real user-space destination(s) while no other threads are running, eliminating the race.

Figure 1 illustrates recording a simple `read` system call. The gray box represents kernel code.

During replay, when the next event to be replayed is an intercepted system call, we set a temporary breakpoint at the address of the system call instruction (recorded in the trace). We use `ptrace` to run the tracee thread until it hits the breakpoint, remove the breakpoint, advance the program counter past the system call instruction, and apply the recorded register and memory changes. This approach is simple and minimizes the number of context switches between RR and the tracee thread. (Occasionally it is unsafe and we fall back to a more complicated mechanism.)

Some system calls manipulate threads or address spaces and require special handling during replay. For example a recorded `mmap` is replayed with `MAP_FIXED` to ensure the mapping is created at the correct address.

2.4 Asynchronous Events

We need to support two kinds of asynchronous events: preemptive context switches and signals. We treat the former as a special case of the latter, forcing a context switch by sending a signal to a running tracee thread. We need to ensure that during replay, a signal is delivered when the program is in exactly the same state as it was when the signal was delivered during recording.

As in previous work [17, 33, 10] we measure application progress using CPU hardware performance counters. Ideally we would count retired instructions leading up to an asynchronous event during recording, and during replay program the CPU to fire an interrupt after that many instructions have been retired — but this approach needs modifications to work in practice.

2.4.1 Nondeterministic Performance Counters

We require that every execution of a given sequence of user-space instructions changes the counter value by an amount that depends only on the instruction sequence, not system state invisible to user space (e.g. the contents of caches, the state of page tables, or speculative CPU state). This property (commonly described as "determinism" [40]) does not hold for most CPU performance counters in practice [17, 40]. For example it does not hold for any "instructions retired" counter on any known x86 CPU model (e.g. because an instruction triggering a page fault is restarted and counted twice).

Fortunately, modern Intel CPUs have exactly one deterministic performance counter: "retired conditional branches" ("RCB"), so we use that. We cannot just count the number of RCBs during recording and deliver the signal after we have executed that number of RCBs during replay, because the RCB count does not uniquely determine the execution point to deliver the signal at. Therefore we pair the RCB count with the complete state of general-purpose registers (including the program counter) to identify an execution point.

In general that *still* does not uniquely identify an execution point (e.g. consider the infinite loop `label: inc [global_var]; jmp label;`). However, in practice we have found it works reliably; code that returns to the same instruction with no intervening conditional branch must be very rare, and it only matters to RR if an asynchronous event occurs at such an instruction — in which case replay would probably diverge and fail.

2.4.2 Late Interrupt Firing

The other major problem is that, although CPUs can be programmed to fire an interrupt after a specified number of performance events have been observed, the interrupt does not fire immediately. In practice we often observe it firing after dozens more instructions have retired. To compensate for this, during replay, we program the interrupt to trigger some number of events earlier than the actual RCB count we are expecting. Then we set a temporary breakpoint at the program counter value for the state we're trying to reach, and repeatedly run to the breakpoint until the RCB count and the general-purpose register values match their recorded values.

2.5 Shared Memory

By scheduling only one thread at a time, RR avoids issues with races on shared memory as long as that memory is written only by tracee threads. It is possible for recorded processes to share memory with other processes, and even kernel device drivers, where that non-recorded code can perform writes that race with accesses by tracee threads. Fortunately, this is rare for applications running in common Linux desktop environments, occurring in only four common cases: applications sharing memory with the PulseAudio daemon, applications sharing memory with the X server, applications sharing memory with kernel graphics drivers and GPUs, and `vdso` syscalls. We avoid the first three problems by automatically disabling use of shared memory with PulseAudio and X (falling back to a socket transport in both cases), and disabling direct access to the GPU from applications.

`vdso` syscalls are a Linux optimization that implements some common read-only system calls (e.g. `gettimeofday`) entirely in user space, partly by reading memory shared with the kernel and updated asynchronously by the kernel. We disable `vdso` syscalls by patching their user-space implementations to perform the equivalent real system call instead.

Applications could still share memory with non-recorded processes in problematic ways, though this is rare in practice and can often be solved just by enlarging the scope of the group of processes recorded by RR.

2.6 Nondeterministic Instructions

Almost all CPU instructions are deterministic, but some are not. One common nondeterministic x86 instruction is `RDTSC`, which reads a time-stamp counter. This particular instruction is easy to handle, since the CPU can be configured to trap on an `RDTSC` and Linux exposes this via a `prctl` API, so we can trap, emulate and record each `RDTSC`.

Other relatively recent x86 instructions are harder to handle. `RDRAND` generates random numbers and hopefully is not deterministic. We have only encountered it being used in one place in GNU `libstdc++`, so RR patches that explicitly. `XBEGIN` and associated instructions support hardware transactional memory. These are nondeterministic from the point of view of user space, since a hardware transaction can succeed or fail depending on CPU cache state. Fortunately so far we have only found these being used by the system `pthreads` library, and we dynamically apply custom patches to that library to disable use of hardware transactions.

The `CPUID` instruction is mostly deterministic, but one of its features returns the index of the running core, which affects behavior deep in `glibc` and can change as the kernel migrates a process between cores. We use the Linux `sched_setaffinity` API to force all tracee threads to run on a particular fixed core, and also force them to run on that core during replay.

We could easily avoid most of these issues in well-behaved programs if we could just trap-and-emulate the `CPUID` instruction, since then we could mask off the feature bits indicating support for `RDRAND`, hardware transactions, etc. Modern Intel CPUs support this ("`CPUID` faulting"); we are in the process of adding an API for this to Linux.

2.7 Reducing Trace Sizes

For many applications the bulk of their input is memory-mapped files, mainly executable code. Copying all executables and libraries to the recorded trace on every execution would impose significant time and space overhead. RR creates hard links to memory-mapped executable files instead of copying them; as long as a system update or recompile replaces executables with new files, instead of writing to the existing files, the links retain the old file data. This works well in practice.

Even better, modern filesystems such as XFS and Btrfs offer copy-on-write logical copies of files (and even block ranges within files), ideal for our purposes. When a mapped file is on the same filesystem as the recorded trace, and the filesystem supports cloning, RR clones mapped files into the trace. These clone operations are essentially free in time and space, until/unless the original file is modified or deleted.

RR compresses all trace data, other than cloned files and blocks, with the `zlib` "deflate" method.

With these optimizations, in practice trace storage is a non-issue. Section 4.4 presents some results.

2.8 Other Details

Apart from those major issues, many other details are required to build a complete record-and-replay system, too many to mention here. Some system calls (e.g. `execve`) are especially complex to handle. Recording and replaying signal delivery are complex, partly because signal delivery has poorly-documented side effects on user-space memory. Advanced Linux kernel features such as `unshare` (kernel namespaces) and `seccomp` require thoughtful handling. Many of these details are interesting, but they do not impact the overall approach.

3 In-process System-call Interception

The approach described in the previous section works, but overhead is disappointingly high (see Figure 5 below). The core problem is that for every tracee system call, as shown in Figure 1 the tracee performs four context switches: two blocking `ptrace` notifications, each requiring a context switch from the tracee to RR and back. For common system calls such as `gettimeofday` or `read` from cached files, the cost of even a single context switch dwarfs the cost of the system call itself. To significantly reduce overhead, we must avoid context-switches to RR when processing these common system calls.

Therefore, we inject into the recorded process a library that intercepts common system calls, performs the system call without triggering a `ptrace` trap, and records the results to a dedicated buffer shared with RR. RR periodically flushes the buffer to its trace. The concept is simple but there are problems to overcome.

3.1 Intercepting System Calls

A common technique for intercepting system calls in-process is to use dynamic linking to interpose wrapper functions over the C library functions that make system calls. In practice, we have found that method to be insufficient, due to applications making direct system calls, and fragile, due to variations in C libraries, and applications that require their own preloading [37, 3]).

Instead, when the tracee makes a system call, RR is notified via a `ptrace` trap and it tries to rewrite the system-call instruction to call into our interception library. This is tricky because on x86 a system call instruction is two bytes long, but we need to replace it with a five-byte `call` instruction. In practice, frequently executed system call instructions are followed by a few known, fixed instruction sequences; for example, many system call instructions are followed by a `cmpl $0xfffff001,%eax` instruction testing the syscall result. We added five hand-written stubs to our interception

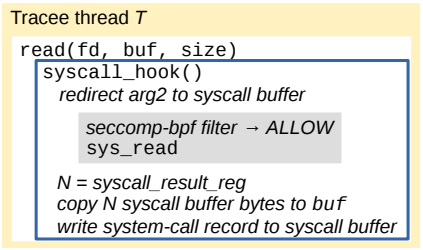


Figure 2: Recording with system-call interception

library that execute post-system-call instructions before returning to the patched code. On receipt of a `ptrace` system-call notification, RR replaces the system call instruction and its following instruction with a call to the corresponding stub.

We (try to) redirect all system call instructions to the interception library, but for simplicity it only contains wrappers for the most common system calls, and for others it falls back to doing a regular `ptrace`-trapping system call.

3.2 Selectively Trapping System Calls

`ptrace` system-call monitoring triggers traps for all system calls, but our interception library needs to avoid traps for selected system calls. Fortunately, modern Linux kernels support selectively generating `ptrace` traps: `seccomp-bpf`. `seccomp-bpf` was designed primarily for sandboxing. A process can apply a `seccomp-bpf` filter function, expressed in bytecode, to another process; then, for every system call performed by the target process, the kernel runs the filter, passing in incoming user-space register values, including the program counter. The filter's result directs the kernel to either allow the system call, fail with a given `errno`, kill the target process, or trigger a `ptrace` trap. Overhead of filter execution is negligible since filters run directly in the kernel and are compiled to native code on most architectures.

Figure 2 illustrates recording a simple `read` system call with in-process system-call interception. The solid-border box represents code in the interception library and the grey box represents kernel code.

RR injects a special page of memory into every tracee process at a fixed address (immediately after `execve`). That page contains a system call instruction — the "untraced instruction". RR applies a `seccomp-bpf` filter to each recorded process that triggers a `ptrace` trap for every system call — except when the program counter is at the untraced instruction, in which case the call is allowed. Whenever the interception library needs to make an untraced system call, it uses that instruction.

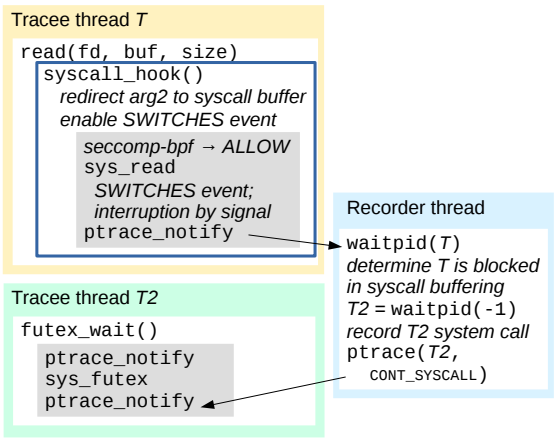


Figure 3: Recording a blocking system call

3.3 Detecting Blocked System Calls

Some common system calls sometimes block (e.g. `read` on an empty pipe). Because RR runs tracee threads one at a time, if a thread enters a blocking system call without notifying RR, it will hang and could cause the entire recording to deadlock (e.g. if another tracee thread is about to `write` to the pipe). We need the kernel to notify RR and suspend the tracee thread whenever an untraced system call blocks, to ensure we can schedule a different tracee thread.

We do this using the Linux `perf` event system to monitor `PERF_COUNT_SW_CONTEXT_SWITCHES`. The kernel raises one of these events every time it deschedules a thread from a CPU core. The interception library monitors these events for each thread and requests that the kernel send a signal to the blocked thread every time the event occurs. These signals trigger `ptrace` notifications to RR while preventing the thread from executing further. To avoid spurious signals (e.g. when the thread is descheduled due to normal timeslice expiration), the event is normally disabled and explicitly enabled during an untraced system call that might block. Still, spurious `SWITCHES` can occur at any point between enabling and disabling the event; we handle these edge cases with careful inspection of the tracee state.

Figure 3 illustrates recording a blocking `read` system call with system-call interception. The kernel deschedules the thread, triggering a `perf` event which sends a signal to the thread, *rescheduling* it, interrupting the system call, and sending a `ptrace` notification to the recorder. The recorder does bookkeeping to note that an intercepted system call was interrupted in thread *T*, then checks whether any tracee threads in blocking system calls have progressed to a system-call exit and generated a `ptrace` notification. In this example *T2* has completed a (not intercepted) blocking `futex` system call, so we resume executing *T2*.

Resuming an intercepted system call that was interrupted by a signal (e.g. *T*'s `read` call in Figure 3) is more complicated. Explaining that requires understanding Linux system call restart semantics, which are too complicated to explain in the space available here.

3.4 Handling Replay

Conceptually, during recording we need to copy system call output buffers to a trace buffer, and during replay we need to copy results from the trace buffer to system call output buffers. This is a problem because the interception library is part of the recording and replay and therefore should execute the same code in both cases. (Previous work with user-level system call interception [11, 36, 21, 29] avoided these problems by having less strict goals for replay fidelity.)

For this reason (and to avoid races of the sort discussed in Section 2.3), the interception library redirects system call outputs to write directly to the trace buffer. After the system call completes, the interception library copies the output data from the trace buffer to the original output buffer(s). During replay the untraced system call instruction is replaced with a no-op, so the system call does not occur; the results are already present in the trace buffer so the post-system-call copy from the trace buffer to the output buffer(s) does what we need.

During recording, each untraced system call sets a result register and the interception library writes it to the trace buffer. Replay must read the result register from the trace buffer instead. We use a conditional move instruction so that control flow is perfectly consistent between recording and replay. The condition is loaded from an `is_replay` global variable, so the register holding the condition is different over a very short span of instructions (and explicity cleared afterwards).

Handling "in-out" system call memory parameters is tricky. During recording we copy the input buffer to the trace buffer, pass the system call a pointer to the trace buffer, then copy the trace buffer contents back to the input buffer. Performing that first copy during replay would overwrite the trace buffer values holding the system call results, so during replay we turn that copy into a no-op using a conditional move to set the source address copy to the destination address.

We could allow replay of the interception library to diverge further from its recorded behavior, but that would have to be done very carefully. We'd have to ensure the RCB count was identical along both paths, and that register values were consistent whenever we exit the interception library or trap to RR within the interception library. It's simplest to minimize the divergence.

3.5 Optimizing Reads With Block Cloning

When an input file is on the same filesystem as the recorded trace and the filesystem supports copy-on-write cloning of file blocks, for large block-aligned `read`s the system call interception code clones the data to a per-thread "cloned-data" trace file, bypassing the normal system-call recording logic. This greatly reduces space and time overhead for file-read-intensive workloads; see the next section.

This optimization works by cloning the input blocks and then reading the input data from the original input file. This opens up a possible race: between the clone and the read, another process could overwrite the input file data, in which case the data read during replay would differ from the data read during recording, causing replay to fail. However, when a file read races with a write under Linux, the reader can receive an arbitrary mix of old and new data, so such behavior would almost certainly be a severe bug, and in practice such bugs do not seem to be common. The race could be avoided by reading from the cloned-data file instead of the original input file, but that performs very poorly because it defeats Linux's readahead optimizations (since the data in the cloned-data file is never available until just before it's needed).

4 Results

4.1 Workloads

Benchmarks were chosen to illuminate RR's strengths and weaknesses, while also containing representatives of real-world usage. They were tuned to fit in system memory (to minimize the impact of I/O on test results), to run for about 30 seconds each (except for *cp* where a 30s run time would require it to not fit in memory).

cp duplicates a `git` checkout of `glibc` (revision 2d02fd07) using `cp -a` (15200 files constituting 732MB of data, according to `du -h`). `cp` is single-threaded, making intensive use of synchronous reads and a variety of other filesystem-related system calls.

make builds DynamoRio [8] (version 6.1.0) with `make -j8` (`-j8` omitted when restricting to a single core). This tests potentially-parallel execution of many short-lived processes.

octane runs the Google Octane benchmark under the Mozilla Spidermonkey Javascript engine (Mercurial revision 9bd900888753). This illustrates performance on CPU-intensive code in a complex language runtime.

htmltest runs the Mozilla Firefox HTML forms tests (Mercurial revision 9bd900888753). The harness is excluded from recording (using `mach mochitest -f plain --debugger RR dom/html/test/forms`). This is an example from real-world usage. About 30% of user-space CPU time is in the harness.

sambatest runs a Samba (git revision 9ee4678b) UDP echo test via `make test TESTS=samba4.echo.udp`. This is an example from real-world usage.

All tests run on a Dell XPS15 laptop with a quad-core Intel Skylake CPU (8 SMT threads), 16GB RAM and a 512GB SSD using Btrfs in Fedora Core 23 Linux.

4.2 Overhead

Table 1 shows the wall-clock run time of various configurations, normalized to the run time of the baseline configuration. *octane* is designed to run for a fixed length of time and report a score, so we report the ratio of the baseline score to the configuration-under-test score — except for replay tests, where the reported score will necessarily be the same as the score during recording. For *octane* replay tests we report the ratio of the baseline score to the recorded score, multiplied by the ratio of replay run time to recording run time. Each test was run six times, discarding the first result and reporting the geometric mean of the other five results. Thus the results represent warm-cache performance.

"Single core" reports the overhead of just restricting all threads to a single core using Linux `taskset`.

"Record no-intercept" and "Replay no-intercept" report overhead with in-process system-call interception disabled (which also disables block cloning). "Record no-cloning" reports overhead with just block cloning disabled.

"DynamoRio-null" reports the overhead of running the tests under the DynamoRio [8] (version 6.1.0) "null tool", to estimate a lower bound for the overhead of using dynamic code instrumentation as an implementation technique. (DynamoRio is reported to be among the fastest dynamic code instrumentation engines.)

4.3 Observations

Overhead on *make* is significantly higher than for the other workloads. Forcing *make* onto a single core imposes major slowdown. Also, *make* forks and execs 2430 processes, mostly short-lived. (The next most prolific workload is *sambatest* with 89.) In-process system-call interception only starts working in a process once the interception library has been loaded, but at least 80 system calls are performed before that completes, so its effectiveness is limited for short-lived processes.

Figure 4 shows the overall recording and replay overhead for workloads other than *make*. Error bars in figures show 95% confidence intervals; these results are highly stable across runs.

Workload	Baseline duration	Record	Replay	Single core	Record no-intercept	Replay no-intercept	Record no-cloning	DynamoRio-null
cp	1.04s	1.49×	0.72×	0.98×	24.53×	15.39×	3.68×	1.24×
make	20.99s	7.85×	11.93×	3.36×	11.32×	14.36×	7.84×	10.97×
octane	32.73s	1.79×	1.56×	1.36×	2.65×	2.43×	1.80×	crash
htmltest	23.74s	1.49×	1.01×	1.07×	4.66×	3.43×	1.50×	14.03×
sambatest	31.75s	1.57×	1.23×	0.95×	2.23×	1.74×	1.57×	1.43×

Table 1: Run-time overhead

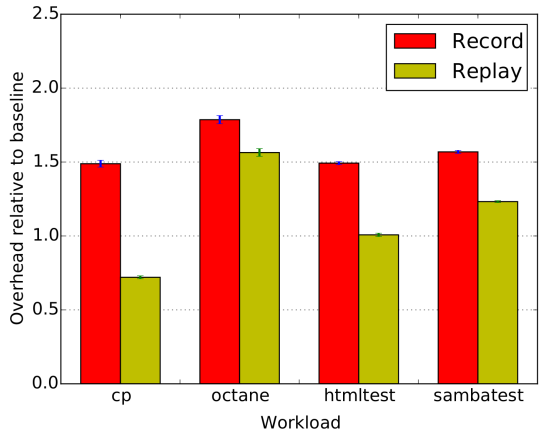


Figure 4: Run-time overhead excluding *make*

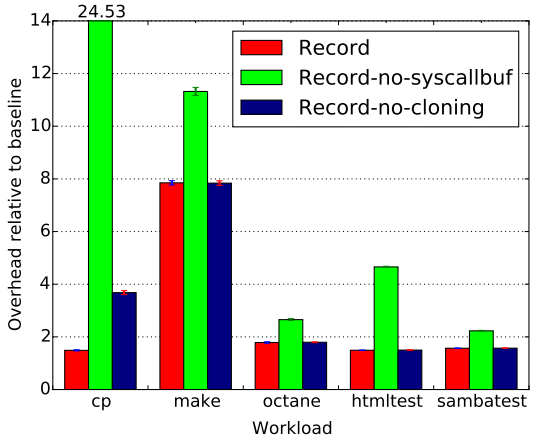


Figure 5: Impact of optimizations

Excluding *make*, RR's recording slowdown is less than a factor of two. Excluding *make*, RR's replay overhead is lower than its recording overhead. Replay can even be faster than normal execution, in *cp* because system calls do less work. For interactive applications, not represented here, replay can take much less time than the original execution because idle periods are eliminated.

octane is the only workload here other than *make* making significant use of multiple cores, and this accounts for the majority of RR's overhead on *octane*.

Figure 5 shows the impact of system-call interception and blocking cloning on recording. The system-call interception optimization produces a large reduction in recording (and replay) overhead. Cloning file data blocks is a major improvement for *cp* recording but has essentially no effect on the other workloads.

Figure 6 compares RR recording overhead with DynamoRio's "null tool", which runs all code through the DynamoRio instrumentation engine but does not modify the code beyond whatever is necessary to maintain supervised execution; this represents a minimal-overhead code instrumentation configuration. DynamoRio crashed on *octane* [1]. *cp* executes very little user-space code and DynamoRio's overhead is low on that workload. On *make* and *sambatest* DynamoRio overhead is similar to RR recording, even though on *make* DynamoRio can utilize multiple cores. On *htmltest* DynamoRio's overhead is very high, possibly because that test runs a lot of Javascript with dynamically generated and modified machine code. Implementing record-and-replay on top of dynamic instrumentation would incur significant additional overhead, so we would expect the resulting system to have significantly higher overhead than RR.

[1] We reported DynamoRio's crash on our "octane" workload to the developers at `https://github.com/DynamoRIO/dynamorio/issues/1930`.

4.4 Storage Space Usage

RR traces contain three kinds of data: cloned (or hard-linked) files used for memory-map operations, cloned file blocks, and all other trace data, especially event metadata and the results of general system calls.

Memory-mapped files are mostly the executables and libraries loaded by tracees. While the original files are not changed or removed, which is usually true in practice, their clones take no additional space and require no data writes. RR makes no attempt to consolidate duplicate file clones, so most traces contain many duplicates and reporting meaningful space usage for these files is both difficult and unimportant in practice. The same is true for cloned file blocks.

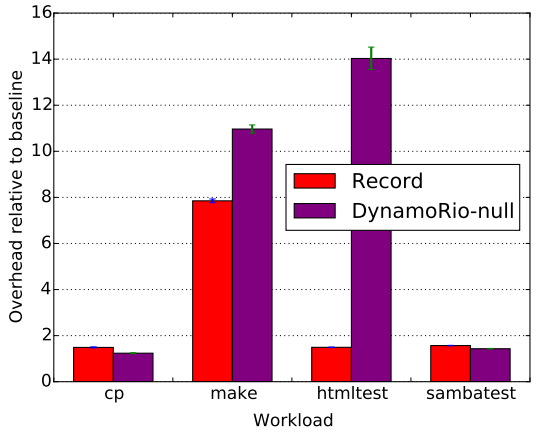


Figure 6: Comparison with DynamoRio-null

Workload	Compressed MB/s	`deflate` ratio	Cloned blocks MB/s
cp	19.03	4.87×	586.14
make	15.82	8.32×	5.50
octane	0.08	8.33×	0.00
htmltest	0.79	5.94×	0.00
sambatest	6.85	21.87×	0.00

Table 2: Storage space usage

Table 2 shows the storage usage of each workload, in MB/s, for general trace data and cloned file blocks. We compute the geometric mean of the data usage for each trace and divide by the run-time of the workload baseline configuration. Space consumption shows very little variation between runs.

Different workloads have highly varying space consumption rates, but several MB/s is easy for modern systems to handle. In real-world usage, trace storage has not been a concern.

4.5 Memory Usage

Table 3 shows the memory usage of each workload. Every 10ms we sum the proportional-set-size ("PSS") values of all workload processes (including RR if running); we determine the peak values for each run and take their geometric mean. In Linux, each page of memory mapped into a process's address space contributes $1/n$ pages to that process's PSS, where n is the number of processes mapping the page; thus it is meaningful to sum PSS values over processes which share memory. The same data are shown in Figure 7. In the figure, the fraction of PSS used by the RR process is shown in orange. Memory usage data was gathered in separate runs from the timing data shown above, to ensure the overhead of gathering

Workload	Baseline	Record	Replay	Single core
cp	0.51	34.54	9.11	0.51
make	510.51	327.19	314.16	288.65
octane	419.47	610.48	588.01	392.95
htmltest	690.81	692.06	324.71	689.75
sambatest	298.68	400.49	428.79	303.03

Table 3: Memory usage (peak PSS MB)

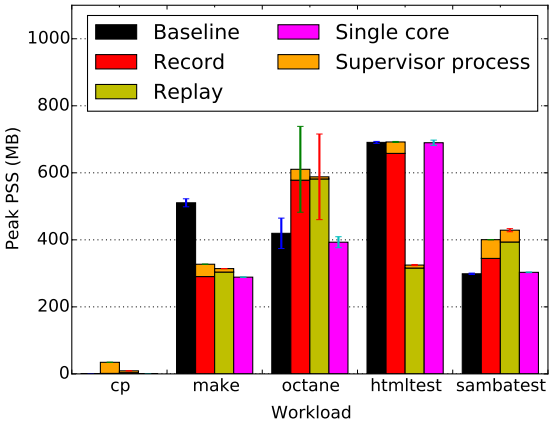


Figure 7: Memory usage

memory statistics did not impact those results.

Given these experiments ran on an otherwise unloaded machine with 16GB RAM and all data fits in cache, none of these workloads experienced any memory pressure. *cp* uses almost no memory. In *make*, just running on a single core reduces peak PSS significantly because not as many processes run simultaneously. In *octane* memory usage is volatile (highly sensitive to small changes in GC behavior) but recording significantly increases application memory usage; recording also increases application memory usage a small amount in *sambatest* but slightly decreases it in *htmltest*. (We expect a small increase in application memory usage due to system-call interception and scratch buffers.) These effects are difficult to explain due to the complexity of the applications, but could be due to changes in timing and/or effects on application or operating system memory management heuristics.

Replay memory usage is similar to recording except in *htmltest*, where it's dramatically lower because we're not replaying the test harness.

RR's memory overhead is not an issue in practice.

5 Hardware/Software Design Constraints

We summarize the hardware and software features RR depends on, for system designers who may be interested in supporting RR-like record-and-replay.

5.1 Hardware

As discussed in Section 2.4.1, RR requires a "determinstic" hardware performance counter to measure application progress. The ideal performance counter for our purposes would count the exact number of instructions retired as observed in user-space (e.g., counting an interrupted-and-restarted instruction once). Virtual machines should support reliable performance-counter virtualization. Currently RR works under KVM and VMware, but VMware's *VM exit clustering* optimization [4], as implemented, breaks the determinism of the RCB counter and must be manually disabled.

Some x86 CPU instructions are nondeterministic. Section 2.6 discusses our current workarounds for this. Exposing hardware support for trapping `CPUID` is important for long-term control over these instructions.

We would like to support record-and-replay of programs using hardware transactional memory (`XBEGIN`/`XEND`). It would suffice if hardware and the OS could be configured to raise a signal on any failed transaction.

Trapping on all other nondetermnistic instructions (e.g. `RDRAND`) would be useful.

Porting RR to ARM failed because all ARM atomic memory operations use the "load-linked/store-conditional" approach, which is inherently nondeterministic. The conditional store can fail because of non-user-space-observable activity, e.g. hardware interrupts, so counts of retired instructions or conditional branches for code performing atomic memory operations are nondeterminstic. These operations are inlined into very many code locations, so it appears patching them is not feasible except via pervasive code instrumentation or compiler changes. On x86(-64), atomic operations (e.g. compare-and-swap) are deterministic in terms of user-space state, so there is no such problem.

5.2 Software

As noted in Section 2.5, RR depends on configuring applications to avoid sharing memory with non-recorded processes.

We described how RR performance depends on modern Linux features: `seccomp-bpf` to selectively trap system calls, `PERF_COUNT_SW_CONTEXT_SWITCHES` performance events to handling blocking system calls, and copy-on-write file and block cloning APIs to reduce I/O overhead.

Efficient record-and-replay depends on clearly identifying a boundary within which code is replayed deterministically, and recording and replaying the timing and contents of all inputs into that boundary. In RR, that boundary is mostly the interface between the kernel and user-space. This suits Linux: most of the Linux user/kernel interface is stable across OS versions, relatively simple and well-documented, and it's easy to count hardware performance events occurring within the boundary (i.e. all user-space events for a specific process). This is less true in other operating systems. For example, in Windows, the user/kernel interface is not publicly documented, and is apparently more complex and less stable than in Linux. Implementing and maintaining the RR approach for Windows would be considerably more challenging than for Linux, at least for anyone other than the OS vendor.

6 Related Work

6.1 Whole-System Replay

ReVirt [17] was an early project that recorded and replayed the execution of an entire virtual machine. VMware [28] used the same approach to support record-and-replay debugging in VMware Workstation, for a time, but discontinued the feature. The full-system simulator Simics supports reverse-execution debugging via deterministic reexecution [20]. There have been efforts to add some record-and-replay support to QEMU [15, 16, 38] and Xen [18, 10]. Whole-system record-and-replay can be useful, but it is often inconvenient to hoist the application into a virtual machine. Many applications of record-and-replay require cheap checkpointing, and checkpointing a VM image is generally more expensive than checkpointing one or a few processes.

6.2 Replaying User-Space With Kernel Support

Scribe [26], dOS [6] and Arnold [14] replay a process or group of processes by extending the OS kernel with record-and-replay functionality. Kernel changes make maintenance and deployment more difficult — unless record-and-replay is integrated into the base OS. But adding invasive new features to the kernel has risks, so if record-and-replay can be well implemented outside the kernel, moving it into the kernel may not be desirable.

6.3 Pure User-Space Replay

Pure user-space record-and-replay systems have existed since at least MEC [11], and later Jockey [36] and liblog [21]. Those systems did not handle asynchronous event timing and other OS features. PinPlay [34], iDNA [7], UndoDB [1] and TotalView ReplayEngine [22] use code instrumentation to record and replay asynchronous event timing. Unlike UndoDB and RR, PinPlay and iDNA instrument all loads, thus supporting parallel recording in

the presence of data races and avoiding having to compute the effects of system calls, but this gives them higher overhead than the other systems. Compared to the other systems that support asynchronous events, RR achieves lower overhead by avoiding code instrumentation.

6.4 Higher-Level Replay

Record-and-replay features have been integrated into language-level virtual machines. DejaVu [12] added record-and-replay capabilities to the Jalapeño Java VM. Microsoft IntelliTrace [2] instruments CLR bytecode to record high-level events and the parameters and results of function calls; it does not produce a full replay. Systems such as Chronon [13] for Java instrument bytecode to collect enough data to provide the appearance of replaying execution, without actually doing a replay. Dolos [9] provides record-and-replay for JS applications in Webkit by recording and replaying nondeterministic inputs to the browser. R2 [23] provides record-and-replay by instrumenting library interfaces; handling data races or asynchronous events requires user effort to isolate the nondeterminism. Such systems are all significantly narrower in scope than the ability to replay general user-space execution.

6.5 Parallel Replay

Recording application threads running concurrently on multiple cores, with the possibility of data races, with low overhead, is extremely challenging. PinPlay [34] and iDNA/Nirvana [7] instrument shared-memory loads and report high overhead. SMP-ReVirt [18] tracks page ownership using hardware page protection and reports high overhead on benchmarks with a lot of sharing. DoublePlay [39] runs two instances of the application and thus has high overhead when the application alone could saturate available cores. ODR [5] has low recording overhead but replay can be extremely expensive and is not guaranteed to reproduce the same program states. Castor [29] instruments synchronization code by modifying compilers and runtime systems, which creates barriers to easy deployment, and cannot replay reliably in the presence of data races.

The best hope for general, low-overhead parallel recording seems to be hardware support. Projects such as FDR [41], BugNet [31], Rerun [24], DeLorean [30] and QuickRec [35] have explored low-overhead parallel recording hardware.

7 Future Work

RR perturbs execution, especially by forcing all threads onto a single core, and therefore can fail to reproduce bugs that manifest outside RR. We have addressed this problem by introducing a "chaos mode" that intelligently adds randomness to scheduling decisions, enabling us to reproduce many more bugs, but that work is beyond the scope of this paper. There are many more opportunities to enhance the recorder to find more bugs.

Putting record-and-replay support in the kernel has performance benefits, e.g. reducing the cost of recording context switches. We may be able to find reusable primitives that can be added to kernels to improve the performance of user-space record-and-replay while being less invasive than a full kernel implementation.

Recording multiple processes running in parallel on multiple cores seems feasible if they do not share memory — or, if they share memory, techniques inspired by SMP-ReVirt [18], dthreads [27] or Castor [29] may work for some workloads.

The applications of record-and-replay are perhaps more interesting and important than the base technology. For example, one can perform high-overhead dynamic analysis during replay [14, 15, 34], potentially parallelized over multiple segments of the execution. With RR's no-instrumentation approach, one could collect performance data such as sampled stacks and performance counter values during recording, and correlate that data with rich analysis generated during replay (e.g. cache simulation). Always-on record-and-replay would make finding and fixing bugs in the field much easier. Demonstrating compelling applications for record-and-replay will build the case for building support into commodity hardware and software.

8 Conclusions

The current state of Linux on commodity x86 CPUs enables single-core user-space record-and-replay with low overhead, without pervasive code instrumentation — but only just. This is fortuitous; we use software and hardware features for purposes they were not designed to serve. It is also a recent development; five years ago `seccomp-bpf` and the Linux file cloning APIs did not exist, and commodity architectures with a deterministic hardware performance counter usable from user-space had only just appeared (Intel Westmere)[2]. By identifying the utility of these features for record-and-replay, we hope that they will be supported by an increasingly broad range of future systems. By providing an open-source, easy-to-deploy, production-ready record-and-replay framework we hope to enable more compelling applications of this technology.

[2]Performance counters have been usable for kernel-implemented replay [17, 33] for longer, because kernel code can observe and compensate for events such as interrupts and page faults.

References

[1] Reversible debugging tools for C/C++ on Linux & Android. http://undo-software.com. Accessed: 2016-04-16.

[2] Understanding IntelliTrace part I: What the @#$% is IntelliTrace? https://blogs.msdn.microsoft.com/zainnab/2013/02/12/understanding-intellitrace-part-i-what-the-is-intellitrace. Accessed: 2016-04-16.

[3] Wine windows-on-posix framework. https://www.winehq.org. Accessed: 2016-09-20.

[4] O. Agesen, J. Mattson, R. Rugina, and J. Sheldon. Software techniques for avoiding hardware virtualization exits. In *Proceedings of the 2012 USENIX Annual Technical Conference*, June 2012.

[5] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, October 2009.

[6] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, October 2010.

[7] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, June 2006.

[8] D. Bruening, Q. Zhao, and S. Amarasinghe. Transparent dynamic instrumentation. In *Proceedings of the 8th International Conference on Virtual Execution Environments*, March 2012.

[9] B. Burg, R. Bailey, A. Ko, and M. Ernst. Interactive record/replay for web application debugging. In *Proceedings of the 26th ACM Symposium on User Interface Software and Technology*, October 2013.

[10] A. Burtsev, D. Johnson, M. Hibler, E. Eide, and J. Regehr. Abstractions for practical virtual machine replay. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, April 2016.

[11] M. E. Chastain. https://lwn.net/1999/0121/a/mec.html, January 1999. Accessed: 2016-04-16.

[12] J.-D. Choi, B. Alpern, T. Ngo, M. Sridharan, and J. Vlissides. A perturbation-free replay platform for cross-optimized multithreaded applications. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, April 2001.

[13] P. Deva. http://chrononsystems.com/blog/design-and-architecture-of-the-chronon-record-0, December 2010. Accessed: 2016-04-16.

[14] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen. Eidetic systems. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, October 2014.

[15] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan. Repeatable reverse engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, December 2015.

[16] P. Dovgalyuk. Deterministic replay of systems execution with multi-target QEMU simulator for dynamic analysis and reverse debugging. 2012.

[17] G. Dunlap, S. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, December 2002.

[18] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, March 2008.

[19] J. Engblom. A review of reverse debugging. In *System, Software, SoC and Silicon Debug Conference*, September 2012.

[20] J. Engblom, D. Aarno, and B. Werner. Full-system simulation from embedded to high-performance systems. In *Processor and System-on-Chip Simulation*, 2010.

[21] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *Proceedings of the 2006 USENIX Annual Technical Conference*, June 2006.

[22] C. Gottbrath. Reverse debugging with the TotalView debugger. In *Cray User Group Conference*, May 2008.

[23] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, December 2008.

[24] D. Hower and M. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, June 2008.

[25] U. Hlzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the 1991 European Conference on Object-Oriented Programming*, July 1991.

[26] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems*, June 2010.

[27] T. Liu, C. Curtsinger, and E. Berger. Dthreads: Efficient deterministic multithreading. In *Proceedings of the ACM SIGOPS 23rd Symposium on Operating Systems Principles*, October 2011.

[28] V. Malyugin, J. Sheldon, G. Venkitachalam, B. Weissman, and M. Xu. ReTrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the Workshop on Modeling, Benchmarking and Simulation*, June 2007.

[29] A. J. Mashtizadeh, T. Garfinkel, D. Terei, D. Mazières, and M. Rosenblum. Towards practical default-on multi-core record/replay. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (to appear)*, April 2017.

[30] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, June 2008.

[31] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[32] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush. Engineering record and replay for deployability: Extended technical report. `http://arxiv.org/abs/1705.05937`.

[33] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2009.

[34] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, April 2010.

[35] G. Pokam, K. Danne, C. Pereira, R. Kassa, T. Kranich, S. Hu, J. Gottschlich, N. Honarmand, N. Dautenhahn, S. King, and J. Torrellas. QuickRec: Prototyping an intel architecture extension for record and replay of multithreaded programs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, June 2013.

[36] Y. Saito. Jockey: A user-space library for record-replay debugging. In *Proceedings of the 6th International Symposium on Automated Analysis-driven Debugging*, September 2005.

[37] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference*, June 2012.

[38] D. Srinivasan and X. Jiang. Time-traveling forensic analysis of VM-based high-interaction honeypots. In *Security and Privacy in Communication Networks: 7th International ICST Conference*, September 2011.

[39] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2011.

[40] V. Weaver, D. Terpstra, and S. Moore. Non-determinism and overcount on modern hardware performance counter implementations. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, April 2013.

[41] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

Improving Storage System Reliability with Proactive Error Prediction

Farzaneh Mahdisoltani
University of Toronto

Ioan Stefanovici
Microsoft Research

Bianca Schroeder
University of Toronto

Abstract

This paper proposes the use of machine learning techniques to make storage systems more reliable in the face of sector errors. Sector errors are partial drive failures, where individual sectors on a drive become unavailable, and occur at a high rate in both hard disk drives and solid state drives. The data in the affected sectors can only be recovered through redundancy in the system (e.g. another drive in the same RAID) and is lost if the error is encountered while the system operates in degraded mode, e.g. during RAID reconstruction.

In this paper, we explore a range of different machine learning techniques and show that sector errors can be predicted ahead of time with high accuracy. Prediction is robust, even when only little training data or only training data for a different drive model is available. We also discuss a number of possible use cases for improving storage system reliability through the use of sector error predictors. We evaluate one such use case in detail: We show that the mean time to detecting errors (and hence the window of vulnerability to data loss) can be greatly reduced by adapting the speed of a scrubber based on error predictions.

1 Introduction

While the storage landscape has changed significantly over the last decade, with hard disk drives (HDDs) and solid state drives (SSDs) each accounting for large shares of the market for persistent storage, one of the key requirements for storage systems has remained the same: store data reliably.

In addition to whole-drive failure, where a drive stops functioning in a way that necessitates replacement, a major threat to storage reliability are *partial* drive failures, where individual sectors on a drive cannot be read. This happens, for example when data in the affected sector is too corrupted to be corrected by drive-internal error correcting codes (ECC). For hard disk drives it can also be due to mechanical damage on the disk surface. The result is the same in either case: the drive cannot recover the data previously stored in the sector.

Field studies show that both solid state drives and hard disk drives experience sector errors at a significant rate. Recent studies based on data from Facebook and Google report that 20-57% of solid state drives experience at least one sector error [15, 22]. A 10-year old study by Bairavasundaram et al. [4] reports that 5–20% of nearline hard disk drives in Netapp's storage systems develop sector errors over a period of 24 months. Our own analysis of recent field data in this paper finds two drive models among the seven most common hard disk models in a large production installation with 11% and 25% of drives affected by sector errors, respectively (see Table 1).

Sector errors are a major concern also when looking into the future. Both SSDs and HDDs continuously grow in capacity to keep up with consumer demand. As there are more sectors on a drive, there is a larger chance that sectors will fail, and as capacity increases, so do drive densities, which can further increase the chances for bit corruption. And some data center operators argue that in order to continue to produce drives at an acceptable price point and with the desired performance characteristics, data center drives should be allowed to have higher error rates and responsibility to deal with those errors should be shifted to higher layers in the storage system [5].

The nature of sector errors makes them challenging to protect against, as they are *latent* errors, i.e. the drive is not aware of and will not report these errors until the affected sector is being accessed. That means storage systems need to proactively periodically read and verify data (a process called disk scrubbing), in order to avoid a situation where a sector error is discovered at a time when it cannot be recovered via redundancy in the system (e.g. during RAID reconstruction).

In this paper we make the case that storage systems would be better prepared to handle sector errors, if errors were predictable. We present techniques for accurately

Drive model	Capacity (TB)	#Drives	Drives (Drive days) affected by:			
			SMART 5	SMART 187	SMART 196	SMART 197
Seagate ST4000DM000	4	36368	1.19% (0.02%)	2.33% (0.01%)	N/A	3.37% (0.02%)
Hitachi HDS5C3030ALA630	3	4664	3.58% (0.05%)	N/A	2.55% (0.04%)	2.72% (0.01%)
HGST HMS5C4040ALE640	3	7168	0.91% (0.03%)	N/A	0.91% (0.03%)	0.59% (0.002%)
Hitachi HDS722020ALA330	2	4774	11.84% (0.12%)	N/A	9.76% (0.08%)	6.47% (0.03%)
HGST HMS5C4040BLE640	4	9426	0.24% (0.003%)	N/A	0.24% (0.003%)	0.32% (0.002%)
Hitachi HDS5C4040ALE630	4	2719	2.54% (0.03%)	N/A	1.62% (0.02%)	1.95% (0.005%)
Seagate ST3000DM001	3	4707	25.15% (1.77%)	30.59% (0.31%)	N/A	35.33% (0.29%)

Table 1: *Overview of HDD models*

predicting errors and show through one specific use case how these predictors can be used to improve storage system reliability. More precisely, we are making the following two contributions:

• We explore a variety of machine learning techniques and show that machine learning models can be trained to predict sector errors with high accuracy. Interestingly, we observe that some of the simplest and easiest to train machine learning models, random forests, are among the most accurate predictors. We also find that the training of predictors is robust, in that even smaller training data sets are sufficient for successful training, and that predictors trained on one drive model can be used to predict errors on a different drive model.

• We propose a number of different use cases for error predictors and explore one of them in depth: improving storage system reliability by adjusting scrub rates based on error prediction. Currently most storage systems run a background scrubber, which at a slow constant speed reads and verifies the stored data to proactively detect errors. Setting the right speed at which to perform scrubbing is tricky, as an overly aggressive scrubber can impact the performance of concurrently running foreground workloads, while a slow scrub speed increases the time it takes to detect an error, hence increasing the system's window of vulnerability to data loss. We propose to adjust scrub speed based on an error predictor, scrubbing faster when errors are predicted and more slowly otherwise. We show that by adjusting the scrub speed based on a predictor the window of vulnerability can be shortened by nearly a factor of 2X, while using the accelerated scrub rate for less than 2% of the total time.

2 A Look at the Field Data

We begin with a description of the field data that our study is based on, including a description of the various error modes and some summary statistics on error frequencies.

2.1 Hard Disk Drives

For our study of errors on hard disk drives and their prediction we use data that has been made publicly available by Backblaze [3]. The entire dataset covers more than a dozen different HDD models and more than a billion device hours, but some models have very small populations. Table 1 provides some summary statistics on the seven drive models with the most data.

The data for these drives is based on SMART (Self-Monitoring, Analysis and Reporting Technology). SMART is a monitoring system supported by most drives that reports on various indicators of drive health, including various types of errors, but also operational data, such as drive temperature, and power on hours of the drive. Backblaze collects for each drive daily snapshots of all SMART values reported by the drive.

To gauge how frequent sector errors are in the drive population at Backblaze, we consider the following SMART parameters that are related to sector errors. Note that the exact definition and reporting of these parameters varies between drive models and manufacturers, and that not all parameters are reported by all drive models.

SMART 5: 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.

SMART 187: The number of read errors that could not be recovered using hardware ECC.

SMART 196: The total count of attempts to transfer data from reallocated sectors to a spare area. Unsuccessful attempts are counted as well as successful.

SMART 197: Count of "unstable" sectors. Some drives mark a sector as "unstable" following a failed read, and remap it only after waiting for a while to see whether the data can be recovered in a subsequent read or when it gets overwritten.

We begin by asking how common sector errors are on the Backblaze drives, since the most recent numbers in the literature [4] are based on data collected more than a decade ago. Table 1 shows, for the most common drive models at Backblaze, the fraction of disk drives and the fraction of drive days that are affected by any of the events corresponding to the 5 SMART parameters above.

We observe that the fraction of drives affected by sector errors is significant: for two of the models 11% and 25%, respectively, of their population have experienced

SSD Model	#Drives	Capacity	Lithography (nm)	PE cycle limit	Avg. PE cycles	Drives (Drive weeks) affected by:	
						Uncorrectable Errors	Bad Blocks
MLC-A	10115	480GB	50	3,000	730	37.07% (0.63%)	50.35% (0.44%)
MLC-B	10151	480GB	43	3,000	949	65.56% (1.09%)	82.75% (1.59%)
MLC-D	10258	480GB	50	3,000	544	46.72% (0.89%)	55.01% (0.57%)

Table 2: *Overview of SSD models*

at least one reallocated sector. We also note that these numbers are significantly higher than the averages reported in previous work [4], which was based on data collected in 2004-2006 in Netapp storage systems and saw 3.45% of drives affected by latent sector errors. However, the numbers we see are in line with those reported for the three nearline drives in the Netapp study, which ranged from 5–20%.

2.2 Solid State Drives

We have been able to obtain data for a randomly sampled subset of around 30,000 drives from three of the four MLC drive models used in a recent field study [22] on SSD reliability based on drives in Google's data centers. We refer to the models as MLC-A, MLC-B, and MLC-D, keeping the naming consistent with that in [22].

The drives in the dataset are based on commodity MLC flash chips, but are custom designed using a custom PCIe interface, firmware and driver. As such, reporting and monitoring is also customized (rather than relying on SMART). For each drive the data contains daily counts for a variety of different types of errors, as well as workload statistics, such as the number of read, write and erase operations during that day. Table 2 summarizes the key statistics for the drives in our data set.

The two events that we are most interested in are uncorrectable errors and bad blocks:

Uncorrectable errors (UEs): A read operations encounters more corrupted bits than the drive-internal ECC can correct. The drive returns an error.

Bad blocks: The drives at Google declare a block bad after an uncorrectable read error, a write error or an erase error, and consequently remap it (i.e. it is removed from future usage and any data that might still be on it and can be recovered is remapped to a different block). Unlike bad blocks for hard disk drives, which refer to disk sectors (typically 512 or 4096 bytes), the blocks here are the unit at which the SSD performs erase operations. The size of an erase block varies with the drive model, but is typically on the order of hundreds of KBs.

3 Predicting Errors

3.1 A Machine Learning Formulation

Our goal is to predict whether a drive will have a sector error within a given time interval, based on its past behavior, as captured by the monitoring data that it reported. We formulate the problem of predicting future errors as a classification problem and then use a variety of methods from machine learning to train classifiers. For simplicity, we assume in the discussion below that we are predicting errors one week into the future, i.e. whether there will be an error within the next 7 days. We create instances (or observations) to our classifier by dividing the data into non-overlapping one-week intervals. For each one-week interval, the response variable (to be predicted) is binary, set either to error or no-error, depending on whether a sector error was observed during this week or not. For explanatory variables (features) we consider all parameters that the drive reports as candidates. The explanatory variables, are based on the monitoring data that the drive produced prior to the prediction interval. More details on the setup of the machine learning formulation follow below.

3.1.1 The Response Variables

As explained in Section 2.1 there are a number of SMART parameters related to sector errors; their exact interpretation can vary between models and not all parameters are reported by all models. We believe that SMART 5 (S5) is the most interesting choice as a response variable, as it is consistently reported by all drive models and because it comprises all the different scenarios that might have led to declaring a sector bad (e.g. independently of how the bad sector was discovered), as it refers to the total number of sectors that have been reallocated. However, we also experiment with training classifiers to predict S187 (read errors that could not be recovered using ECC) and S197 (sectors became unstable).

For the SSD data choosing the response variables is straightforward, since all drive models use the same customized reporting mechanisms. We experiment with two different response variables: uncorrectable errors and bad blocks.

3.1.2 The Explanatory Variables

For the HDD data we consider all SMART parameters reported by a drive as possible candidates for explanatory variables. We convert the raw data into explanatory variables in two ways. The first set of input variables consists of the most recent raw values of all the parame-

ID#	Attribute Name
*S*1	Read Error Rate
*S*4	Start/Stop Count
*S*5	Reallocated Sectors Count
*S*7	Seek Error Rate
*S*9	Power-On Hours (POH)
*S*12	Power Cycle Count
*S*187	Reported Uncorrectable Errors
*S*193	Load Cycle Count
*S*194	Temperature
*S*197	Current Pending Sector Count
*S*199	UltraDMA CRC Error Count

Table 3: *SMART attributes selected as learning features for HDD devices*

ters reported by the drive before the beginning of the one-week interval. We also add a second set of input variables based on transformations of some smart parameters. In particular, some SMART parameters are reported as cumulative counts over a drive's lifetime (e.g. SMART 189 contains the total number of high fly writes a disk has ever experienced). However, for predicting errors during a given time period, it might be more insightful to know the recent rate of change for this variable, rather than the cumulative lifetime count. We therefore include for each cumulative smart parameter a second input variable that consists of the *increase* in value that the corresponding smart parameter experienced during the one-week window into the past.

As the resulting number of input variables is large, we perform feature selection before training machine learning models on the data. We use correlation coefficients and information gain to determine the features that correlate most with the drive errors and we remove the SMART attributes that never change for a specific drive model. Table 3 lists the SMART attributes we used to build prediction models for HDDs. For a more detailed explanation of the various fields see an overview of all SMART parameters [23]. Table 4 provides the attributes used for SSDs. For a more detailed explanation of the various fields, see [22].

3.1.3 Training the Classifiers

We experiment with five different machine learning methods that are commonly used for classification problems: classification and regression trees (CART), random forests, support vector machines, neural networks and logistic regression. For random forests we experimented with different numbers of trees, and settled on using 20 trees for the results included in the paper. We ran experiments with up to 100 trees, but did not see significant improvements. For support vector machines, we experiment with three different kernels: polynomial, linear and radial basis function (RBF) kernels. We also experimented with different degrees for the polynomial kernels. For neural networks, we include results for a network with 3 layers and 100 nodes. Neural networks with larger numbers of layers are impractical for learning rare events (such as errors or failures) as they require massive amounts of training data. We also experimented with more advanced type of neural networks, such as recurrent neural networks, but didn't find the results to improve upon standard neural networks, and hence chose not to include the results. We use the hold-out method to find the best values to adapt the parameters of neural networks, including learning rate, momentum and regularization factors. We perform a grid search on these parameters to find the combination that achieves the highest performance. For logistic regression we experimented with different values for regularization and learning rate. All methods were implemented in Matlab. For SVM we used the LIBSVM library [6].

ID#	Attribute Name
1	correctable error
2	erase count
3	erase error
4	factory bad block
5	final read error
6	final write error
7	meta error
8	read count
9	read error
10	response error
11	status dead
12	status read only
13	timeout error
14	timestamp usec
15	uncorrectable error
16	write count
17	write error
18	cumulative bad block count fixed
19	weekly bad block count
20	cumulative pe cycle fixed
21	weekly pe cycle

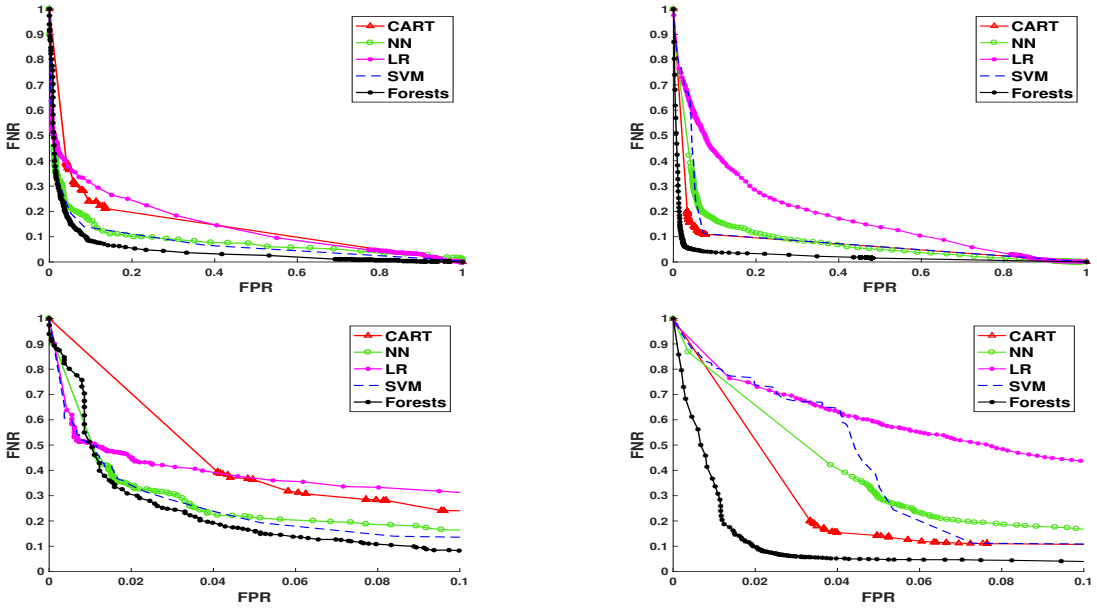
Table 4: *Attributes selected as learning features for SSD devices*

As the range of values spanned by different features varies widely, we employed data normalization using the feature scaling method to avoid bias towards features with larger parameter values. Feature scaling transforms each attribute in the data using the following formula:

$$X' = \frac{X - X_{min}}{X_{max} - X_{min}} \quad (1)$$

where X is the original value of a feature. X_{max} and X_{min} are respectively the maximum value and the minimum value of this feature for the subset of data for each disk manufacturer and model.

When creating the training data sets, we use majority class under-sampling, a standard technique to improve training in the case of very imbalanced classes, which arises because the original data set has many more in-



(a) Hitachi (b) Seagate

Figure 1: *False positive rates (x-axis) versus false negative rates (y-axis) when predicting sector reallocation (SMART 5) for Hitachi and Seagate HDDs. The bottom row shows a close-up of the false positive range* $[0;0.1]$ *on the x-axis.*

stances of negatives (no error), than positives (error). E.g. if we randomly selected training instances from the entire data set, the training set would include only a very small number of instances with errors and hence bias the training process towards non-error predictions. Instead we undersample with different ratios, making sure that at least a certain fraction of training instances are error instances. We experimented with different ratios ranging from 1:1 to 1:10, and found that most ratios, where about 20-60% of all training instances are positives, work well. We have chosen to include the results for a ratio of 1:3.

When performing the training, we divide the data into 75% training data and 25% testing data for most experiments, as is common practice in the machine learning community. Later in the paper, we will also show (Section 3.3) that much smaller training sets are sufficient. We used the hold-out method for tuning parameters of different algorithms, and chose the values which led to the highest quality predictions.

3.1.4 Metrics

We measure the success of different machine learning models by reporting two standard measures: the false positive rate (FPR) and the false negative rate (FNR).

The false positive rate measures the frequency of false alarms. It is the fraction of time intervals that did not experience an error, but was falsely predicted to have an error:

$$\text{FPR} = \frac{\#\text{false positives}}{\#\text{false positives} + \#\text{true negatives}}$$

The false negative rate measures what fraction of errors was missed, i.e. the fraction of intervals that had an error, but was predicted not to have an error:

$$\text{FNR} = \frac{\#\text{false negatives}}{\#\text{false negatives} + \#\text{true positives}}$$

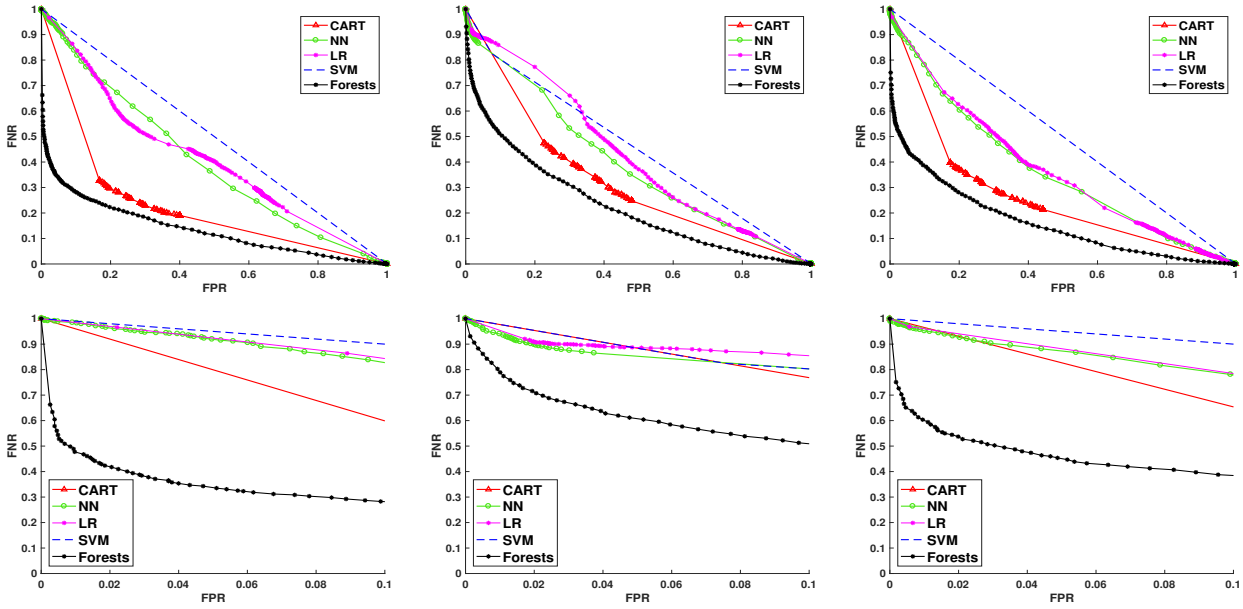
3.2 Prediction Results

3.2.1 Prediction Results for HDDs

We train classifiers to predict SMART 5 (sector reallocations) for two of the hard disk drive models, Hitachi's HDS722020ALA330 and Seagate's ST3000DM001. We chose those two drive models to cover two different manufacturers, because they are among the most common drives in the HDD dataset and because they are the drive models with the highest error rates.

The graphs in Figure 1 summarize the quality of the predictions we obtain using the various machine learning methods on each of the two device types. The top row shows results for the entire range of false positive ratios (x-axis), while the bottom row shows a close-up of the x-axis range with false positive ratios less than 0.1.

We make several observations. First, errors can be predicted with a high accuracy. For example, when limiting the false positive rate (i.e. the rate of false alarms) to 10% we can correctly predict 90% and 95% of all errors for Hitachi and Seagate, respectively. When limiting the false positive rate more conservatively to 2% we can still correctly predict 70-90% of the errors.



(a) MLC-A (b) MLC-B (c) MLC-D

Figure 2: *False positive rates (x-axis) versus false negative rates (y-axis) when predicting uncorrectable errors for the three SSD models. The bottom row shows a close-up of the false positive range* $[0;0.1]$ *on the x-axis.*

To put those false positive rates into context, recall that our goal is to use predictions to proactively trigger light-weight protection mechanisms, such as a data scrub to proactively detect errors. So, while these false positive rates would likely be too high in the context of predicting whole-disk failures and triggering drive replacements, they are acceptable in our context.

Since both drive models report SMART 197 (number of unstable sectors), one might wonder whether the predicted sector reallocations are trivial predictions based on sectors that were already previously known to be unstable. We verify that this is not the case by removing SMART 197 as an input variable, retraining the classifier and still achieving the same results. We also observe no correlation between the SMART 5 and SMART 197, based on correlation coefficients.

When comparing different machine learning methods, we observe that random forests consistently outperform or match the performance of other classifiers. This is encouraging for use in practice as random forests are among the classifiers that are the easiest and fastest to train, as there are very few parameters to tune. The main parameter is the number of trees in the forest, and we find that the results are not very sensitive to this parameter. For example, we find that results are the same for forests with 20, 50, 100 and 200 trees.

On the other hand, some of the other classifiers, in particular neural networks, support vector machines and logistic regression required a considerable amount of tuning as part of a lengthy training process. Despite our extensive efforts in training these models, their performance can only barely and only for small false positive ranges match that of random forests.

Finally, we also repeated training and prediction for two other SMART parameters, S187 and S197. The results are included in the appendix. We see similar trends as for S5, in that random forests match or outperform other predictors and we find that prediction accuracy is high, albeit slightly lower than for S5. We hypothesize that the accuracy for S187 and S197 is slightly lower, because whether a bad sector will affect the counts for these two variables will depend on how the bad sector was discovered (e.g. by a read or a write operation), which is a somewhat random factor that would be hard to predict.

3.2.2 Prediction Results for SSDs

In this section we train classifiers to predict uncorrectable errors for the three types of SSDs that we have data for. Figure 2 shows the results. We observe that, as was the case for HDDs, random forests outperform other classifiers. Sector errors can be predicted with a significant accuracy, albeit somewhat lower accuracy than for HDDs. At a false positive rate of 10% the random forest classifier catches 50-70% of errors. At a false positive rate of 2% the classifier can still catch 50-60% of errors for two of the three models.

We also experimented with training classifiers to predict bad blocks and include the results in the appendix.

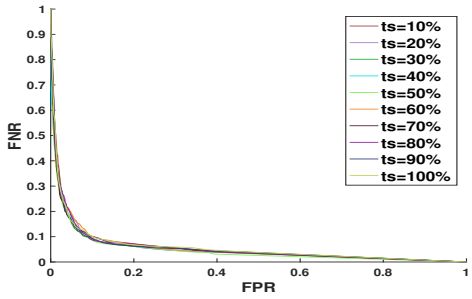


Figure 3: *Training of the random forest classifier for the Hitachi drive with only a fraction of the original training set still leads to nearly identical quality of predictions.*

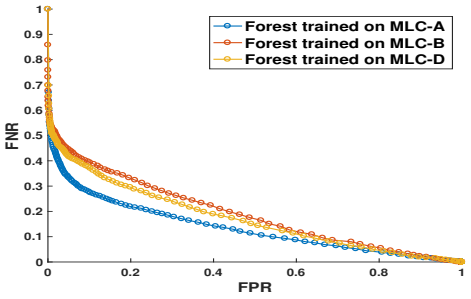
Random forests are again the classifier with the best performance, however prediction accuracy is lower than for uncorrectable errors. A block can be declared due to an uncorrectable read on it, or when (even after a number of retries) write or erase operations fail. Our results seem to indicate that failing write or erase operations are harder to predict than uncorrectable errors.

3.3 Robustness of Predictions

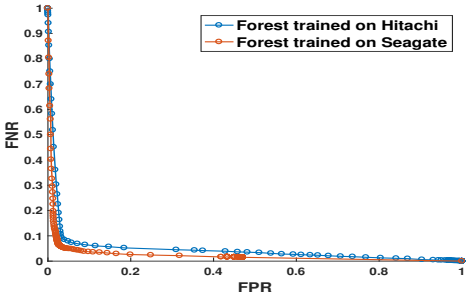
Our predictions in the previous section were based on a relatively large amount of data. One problem in practice is that an operator might not have access to data sets of comparable size (e.g. because the system is smaller or still relatively new, and hence not much data is available yet). To address this problem, we experimented with two different approaches. We repeated the training process, but with significantly smaller amounts of training data. Figure 3 shows the results, when training random forests for the Hitachi HDD on only a fraction of the original training set, ranging from only 10%, to 90% of the training data that was used in the previous section. We observe that prediction quality is hardly affected.

We also experimented with solutions to the problem that no prior data is available when a new type of device is first deployed. In particular, we ran experiments where we use a predictor that was trained on one drive model to make prediction for another drive model. Figure 4 shows the accuracy of predictions when we use random forests trained on MLC-B and MLC-D to predict errors for MLC-A (Figure 4(a)) and we use random forests trained on the Hitachi drive to predict sector errors on the Seagate drive (Figure 4(b)). We observe that while prediction accuracy drops, the quality of the resulting predictions is still high.

This observation is particularly interesting as the drives that were used for model building differed in many important aspects significantly from those we make predictions for. For example, SSD model MLC-B differs from MLC-A in manufacturer and lithography and the



(a) MLC-A



(b) Seagate

Figure 4: *Results for cross-model prediction: We use random forests trained on MLC-B and MLC-D to predict errors for MLC-A in the top figure and we use random forests trained on the Hitachi drive to predict sector errors on the Seagate drive in the bottom figure.*

two drive models have vastly different rates of uncorrectable errors. The two hard disk drives differ in their manufacturer, their capacity and their rate of sector errors. Moreover, the data for the two hard disk drives is based on SMART reporting, which is inconsistent between different manufacturers and models. For example, the Hitachi model does not report all the SMART parameters the Seagate model reports.

4 Tuning Scrub Rates based on Predictions

In the previous section we have developed classifiers to predict future sector errors. In this section, we explore one particular idea for how such predictions could be used to improve the reliability of storage systems.

4.1 Idea

Most storage systems employ a data scrubber to protect against data loss due to sector errors. A scrubber is a background process that periodically performs full-disk scans to proactively detect and correct sector errors. For example, some filesystems, such as ZFS and Btrfs, provide scrubbing at the filesystem level, RAID controllers may initiate periodic scrubs at the block level, and commercial storage systems, such as Netapp's, often support

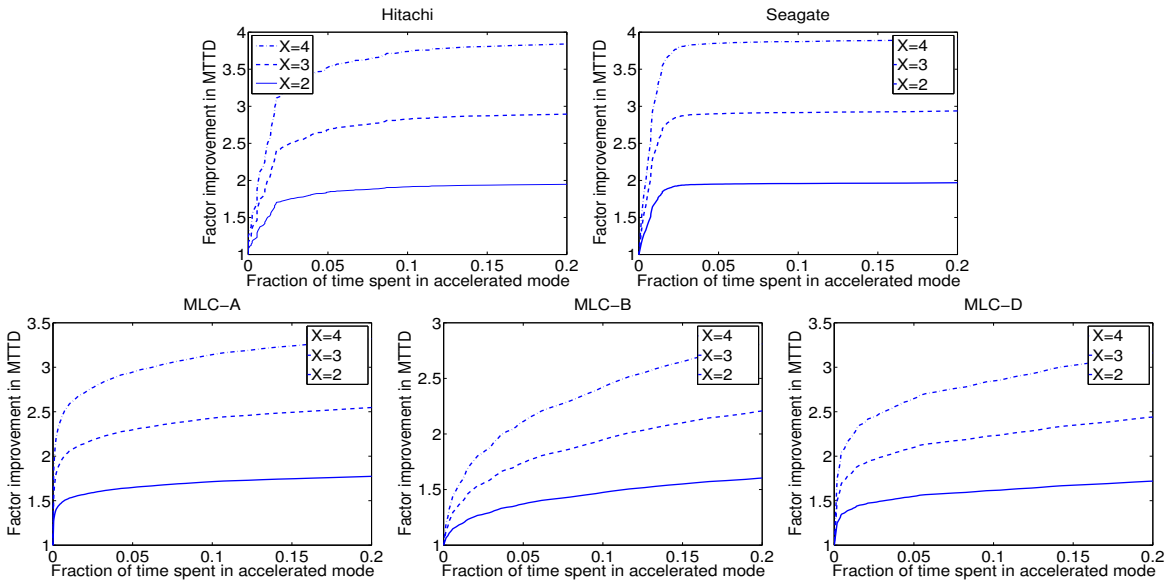


Figure 5: *Simulating the adaptive scrubber. The X-axis shows the fraction of time the scrubber spends in accelerated mode and the Y-axis shows the factor decrease in mean time to error detection. Each line corresponds to a different factor X of acceleration.*

scrubbing at the file and the block level (termed data scrub versus media scrub, respectively [4]).

The goal of the scrubber is to minimize the time between the occurrence of an error and its detection (Mean Time To Detection, MTTD), since during this time the system is vulnerable to data loss (e.g. if a RAID array experiences a whole-disk failure(s) before the sector error is detected and corrected). In addition to minimizing MTTD, a scrubber must ensure that it does not significantly affect the performance of foreground workloads running on the system.

Currently, administrators in practice configure a scrubber to run at a *fixed* scrub rate, which must balance the two goals above: Scrubbing at a fast rate will detect errors more quickly, while a slow scrub speed imposes less load on the system. A common rule of thumb is to complete one full scrub of a drive once a week or bi-weekly.

Instead, we propose to *adapt* the scrub rate dynamically based on the predicted chance of encountering errors, rather than using one fixed scrub rate throughout. This is similar to an idea proposed as future work by Ma et al. [13], who suggest to increase the scrub rate for drives with higher error rates. We use our methods from Section 3.2 to predict errors, and whenever an error is predicted we accelerate the speed at which the system scrubs. When no error is predicted we reduce the scrub speed. Note that errors are rare events, so provided we choose a predictor with a reasonably low rate of false positives, the system should rarely trigger accelerated scrubs.

4.2 Evaluation Setup

We set up a series of simulations to evaluate the effectiveness of an adaptive scrubber compared to a fixed rate scrubber. We consider adaptive scrubbers that switch between two speeds, s_1 and s_2, where s_1 is the slower speed that is used when no errors are predicted and s_2 is the accelerated speed that is used when an error is predicted. One might also consider a continuous spectrum of scrub speeds, but we defer this discussion to Section 4.4.

In all simulations, we rely on predictors based on random forests, as they tended to provide the highest quality predictions. The training of a random forest for a given device type can be configured to achieve different rates of false positives and false negatives. The rate of false positives that is acceptable, will be system dependent based on the system's sensitivity to added workload. For example, for a false positive rate of 2% a system will spend roughly 2 weeks out of the 52 weeks in a year scrubbing at an accelerated rate, without catching any errors. At a lower false positive rate, the system will spend less time in accelerated scrub mode, but will also be slower at catching some errors (as the false negative rate will go up). We therefore experiment with a range of different configurations for the random forest.

We set the s_1 parameter, the default scrub rate when no errors are predicted, to one full disk scrub per week, as this is a common scrub frequency in practice. The second parameter is s_2, i.e. the rate at which the system scrubs in accelerated mode. Again the choice of this parameter will depend on the system's sensitivity to

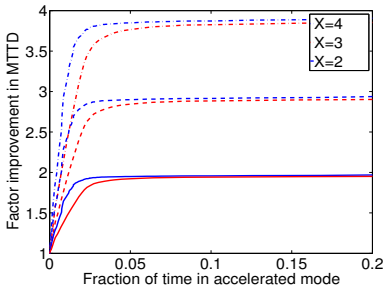


Figure 6: *Results when the adaptive scrubber employs the classifier trained for the Hitachi drive to predict errors on the Seagate drive (red lines) compared to scrub results when prediction for Seagate is done on the models trained on Seagate data (blue lines).*

added workload. If we increase the scrub speed by a factor of X during an accelerated scrub, the MTTD will on average be reduced by a factor of X, but the system experiences a higher load. In our simulations, we therefore experiment with a range of X values. We perform predictions once a week.

4.3 Results

Figure 5 shows the results when simulating the adaptive scrubber using random forests. The X-axis shows the fraction of time the scrubber spends in accelerated mode and the Y-axis shows the factor decrease in mean time to error detection. Each line corresponds to a different factor of acceleration used by the scrubber when an error is predicted. E.g. a 2X factor of acceleration means that the scrub rate is doubled, i.e. if the default scrub rate is to scrub once per week, in accelerated mode the scrubber will complete two scrubs per week.

We observe that even if we limit the time the system is spending in the accelerated scrub mode to 2% of the total time (i.e. we are using predictors with a very low false positive rate), the factor decrease in mean time to error detection is significant. For example, in the case of the hard disk drives when the scrub speed is doubled upon an error prediction we detect errors on average 1.7-1.8X faster than a fixed rate scrubber. Even for the SSDs, for which the classifiers' predictions were less accurate, the savings are still significant. E.g. for MLC-A and MLC-D errors are detected on average 1.4-1.5X faster when the scrub speed is doubled upon an error prediction, even if the total time spent in accelerated mode is limited to 2%.

We also consider the case where predictors for a given drive model are not available, e.g. because the drive model is new and has just been deployed and where predictions are obtained by using classifiers trained for a different drive model. Recall that in Section 3.3 we predicted errors for the Seagate model based on a classifier trained for the Hitachi model. Figure 6 shows the results when the adaptive scrubber adjusts scrub speeds for the Seagate drive based on predictions from the classifier trained on the Hitachi drive. We observe that the improvements in the mean time to detection are still very good.

4.4 Refinements and Practical Concerns

While our results provide a proof-of-concept for prediction-based tuning of scrub rates, the methods could be further refined. For example, we only considered a scrubber that alternates between two speeds. One might further improve results by adjusting scrub speeds on a continuous spectrum, based on the certainty of an error prediction. In their raw form forests (and most of the other methods we considered) produce error probabilities, rather than a binary error versus no-error prediction. The binary predictions are produced by applying some thresholds to the produced probabilities. One could instead use the raw probabilities and adjust the scrub speed along a continuous spectrum based on how large the current error probability is, and might do better than in the simple two-speed scrubber.

One practical concern with accelerating scrub speeds is the impact that the additional load has on the system. There are tools available to an administrator to mitigate possible negative effects on foreground workload. Besides the obvious one of limiting the increase in scrub speed to some maximum value the administrator deems tolerable for their system, an administrator could also put a limit on the maximum amount of time the system spends in accelerated scrubs. Moreover, there are techniques that have been suggested recently [1] to reduce the impact of storage maintenance workloads on the system that can be applied to scrubbing as well.

A secondary concern might be that the additional load imposed on the system might induce new errors. While solid state drives are known to wear out faster under heavy write workloads, scrub operations consist of reads only, and two recent papers [15, 22] independently show that there is no correlation between the number of reads and the number uncorrectable errors in a system. Similarly, for hard disk drives one might expect writes to have a correlation with sector errors, as an incorrect write (e.g. a high-fly write) might be the cause of sector errors, but a study of field data [21] finds no correlation between read operations and sector errors.

5 Other Use Cases for Error Prediction

This section proposes and discusses the use of error predictors as adaptive policy-enforcing mechanisms in storage systems. We leave the detailed exploration of these use cases to future work.

5.1 Tuning Drive Internal Mechanisms

5.1.1 Adaptive Error Correcting Codes

There has been some recent work in the flash community to equip drives with adaptive error correcting codes [7, 10]. The original motivation was that the reliability of flash cells changes over their lifetime, so one could use a smaller, less powerful code at the beginning of a drive's life and switch to larger, but stronger codes later on, as the drive ages. Rather than using the age of a drive as an indicator whether the drive should switch to the stronger ECC, it would be natural to use an error predictor, and trigger the switch when predictions indicate that the drive is more likely to develop uncorrectable errors.

5.1.2 Proactive Retirement of Blocks or Chips

Our predictors only predict whether a drive will experience sector errors or bad blocks, or not. We do not predict the location of any future errors (i.e. which block and which chip), because the field data available to us does not include location information. However, it is likely that errors occur at or near those locations that are responsible for early symptoms of errors (e.g. various types of prior errors).

It would be interesting to also predict the precise location of future errors, and investigate whether these predictions can be used to proactively retire a block or a chip. We hope this work will encourage storage manufacturers and others to explore making such predictions using data available inside drives, and exposing this information to consumers.

5.2 Tuning the Cache Policy in SSD Caches

SSDs are not only used for persistent storage of data, but also as a caching layer. In the case of write-through caches, errors do not pose a risk for data loss. They just translate to higher read latency for data affected by errors, as accessing it will turn into a cache miss. In the case of write-back caches, errors create the potential for data loss if they affected dirty data in the cache that has not yet been flushed to persistent storage. One could therefore consider a policy where an error predictor is used to switch the cache policy from write-back to write-through when a predictor indicates future errors.

5.3 Tuning Filesystem Mechanisms

Many filesystems contain mechanisms to protect against sector errors. These range from replicating important data structures (such as a filesystem's super-block), to adding checksums for metadata (such as inodes), replicating data as well as metadata, or even RAID-5 or RAID-6 level data protection in the case of ZFS. An interesting area of future work is to explore which of these mechanisms could be dynamically enabled using error predictors. This would be particularly useful if filesystem mechanisms such as metadata replication had some advance warning about which specific blocks or chips were likely to fail (as discussed in Section 5.1.2).

6 Related Work

6.1 Predicting Errors in Storage Systems

To the best of our knowledge there is no prior work on predicting *partial* drive failures, such as sector errors on hard disks and uncorrectable errors and bad blocks in solid state drives. Instead prior work on drive reliability predictions focuses on predicting *complete* drive failure for HDDs [8, 9, 11–13, 16, 24] and SSDs [17].

The motivations and requirements for predicting complete drive failure are very different from our work on predicting partial drive failures. The goal is to use predictions of drive failures in order to initiate proactive drive replacement before the failure occurs. Such predictions require an extremely low false positive rate, since the unnecessary replacement of healthy drives is associated with significant costs. In contrast, we are interested in predicting partial drive failures to guide the system to take lighter-weight proactive measures, such as increasing the scrub rate. Another difference between our work and much of the prior work is that training and test instances in prior work were often not drives deployed in productions systems, but rather drives run in a controlled environment by the manufacturer [11,16], or the data sets that were used were of very limited size (e.g. less than 20 faulty drives in [8, 9]).

Finally, our work explores a wide range of modern machine learning techniques, which subsume most of the techniques used in prior work. Hamerly et al. [9] use Bayesian approaches. We applied logistic regression instead, as naive Bayes methods make a strong assumption on the conditional independence of the input variables. Hughes et al. [11] use statistical hypothesis tests, but find in later work that SVMs perform better [16]. Our work includes SVMs, as well as a number of other techniques not explored by [11,16]. The work by Pinheiro et al. [20] does not attempt to predict drive failures, however they observe a correlation between some SMART parameters and drive failures. The work by Ma et al. [13] uses a simple threshold-based prediction. They find that by putting a threshold on the number of sector reallocations drive failure can be predicted well. We experimented with this approach and find that it performs poorly on the hard disk data, compared to the machine learning techniques, and is limited to results with very high false negative rates for the solid state drives.[1]

[1] For space reasons we do not include detailed results for thresholding in this work, but instead refer the reader to a tech-report [14].

We are aware of only three papers that use machine-learning based techniques on large-scale field data and again they are predicting whole drive failures [12,17,24]. Two of the papers focus on HDDs and find that CART models outperform SVMs and neural networks [12, 24]. Narayanan et al. [17] show that random forests can predict fail-stop events in SSDs (events that lead to server shut-down and typically drive replacement), albeit accuracy is lower than that reported in papers predicting HDD failure. Our work includes all these methods, among others, and we find that forests are superior to the other methods for predicting sector errors.

6.2 Improving Scrubbers

Prior work on scrubbing is mostly focused on minimizing the impact of scrub operations on foreground traffic, e.g. by trying to submit scrub operations during idle times [2] or by piggy-backing it on workload operations [1], or reducing the time to detect errors by modifying the order in which sectors are scrubbed [18]. The only work we are aware off that adjusts scrub speeds is by Paris et al. [19] and proposes to perform an expedited scrub run after a whole-disk failure in a RAID-6 array. Our idea of adjusting scrub speed based on error predictions is similar to Ma's [13] future work suggestion of increasing scrub rates for drives with high error rates. Our work on adjusting scrub speed based on error predictions is orthogonal to work on minimizing the impact of scrub operations or the optimal ordering of scrub operations within a scrub and can be used in conjunction with any of these techniques.

7 Sharing of Data

All the data on hard disk drives used in the paper is already publicly available on the Backblaze home page, so all our results are reproducible by other researchers. The data on solid state drives has been shared with us by Google, and we are currently working with our collaborators at Google towards sharing this data publicly.

8 Conclusion

This paper explores the use of machine learning to make storage systems more reliable in the face of latent sector errors. We experiment with a wide variety of machine learning techniques and find that sector errors in hard disk drives and solid state drives can be predicted accurately with classifiers based on random forests, which are easy to train and to parameterize. We show that training is robust even for small training sets or when training data comes from a different drive model than the target system. We also discuss a number of possible use cases for improving storage system reliability through the use of sector error predictors. We evaluate one such use case in detail: We show that the mean time to detecting errors (and hence the window of vulnerability to data loss) can be greatly reduced by adapting the speed of a scrubber based on error predictions.

Appendix

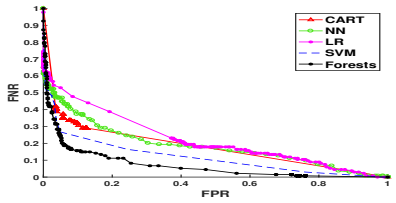


Figure 7: *Predicting SMART 197 for Hitachi*

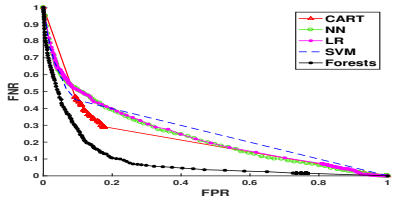


Figure 8: *Predicting SMART 187 for Seagate*

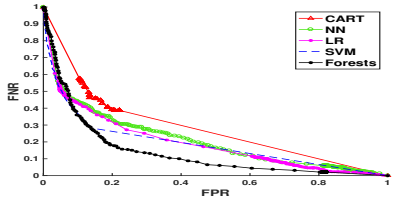


Figure 9: *Predicting SMART 197 for Seagate*

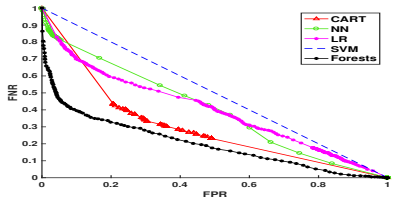


Figure 10: *Predicting bad blocks for MLC-A*

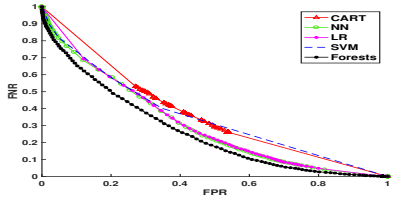


Figure 11: *Predicting bad blocks for MLC-B*

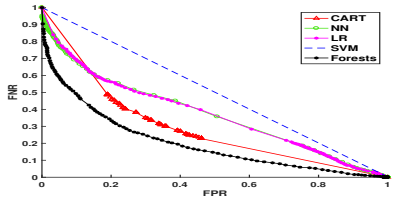


Figure 12: *Predicting bad blocks for MLC-D*

9 Acknowledgments

We would like to thank the Usenix ATC '17 anonymous reviewers, and our shepherd Fred Douglis for the valuable feedback and the many good questions they brought up. While we did not have room to address all of them in this paper, we refer the reader to an extended version of our paper, including additional experiments, which addresses many of these questions [14]. We would also like to thank Arif Merchant for his continued efforts to make the Google SSD data available.

References

[1] AMVROSIADIS, G., BROWN, A. D., AND GOEL, A. Opportunistic storage maintenance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 457–473.

[2] AMVROSIADIS, G., OPREA, A., AND SCHROEDER, B. Practical scrubbing: Getting to the bad sector at the right time. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on* (june 2012), pp. 1 – 12.

[3] BACKBLAZE. The backblaze hard drive data and stats. https://www.backblaze.com/b2/hard-drive-test-data.html, 2016.

[4] 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.

[5] BREWER, E., YING, L., GREENFIELD, L., CYPHER, R., AND TS'O, T. FAST'16 Keynote talk: Disks for Data Centers. Tech. rep., Google, 2016.

[6] CHANG, C.-C., AND LIN, C.-J. The libsvm library for matlab. https://www.csie.ntu.edu.tw/ cjlin/libsvm, 2017.

[7] CHEN, T.-H., HSIAO, Y.-Y., HSING, Y.-T., AND WU, C.-W. An adaptive-rate error correction scheme for nand flash memory. In *VLSI Test Symposium, 2009. VTS'09. 27th IEEE* (2009), IEEE, pp. 53–58.

[8] 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.

[9] HAMERLY, G., ELKAN, C., ET AL. Bayesian approaches to failure prediction for disk drives. In *ICML* (2001), vol. 1, Citeseer, pp. 202–209.

[10] HARATSCH, E. F. Nand flash media management algorithms. In *Flash Memory Summit* (2016).

[11] HUGHES, G. F., MURRAY, J. F., KREUTZ-DELGADO, K., AND ELKAN, C. Improved disk-drive failure warnings. *IEEE Transactions on Reliability 51*, 3 (2002), 350–357.

[12] 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 *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014* (2014), pp. 383–394.

[13] 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. *Trans. Storage 11*, 4 (Nov. 2015), 17:1–17:28.

[14] MAHDISOLTANI, F., STEFANOVICI, I., AND SCHROEDER, B. Improving storage system reliability with proactive error prediction. Tech. Rep. 633, University of Toronto, 2017.

[15] 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.

[16] MURRAY, J., HUGHES, G., AND KREUTZ-DELGADO, K. Machine learning methods for predicting failures in hard drives. *Journal of Machine Learning Research* (2005).

[17] NARAYANAN, I., WANG, D., JEON, M., SHARMA, B., CAULFIELD, L., SIVASUBRAMANIAM, A., CUTLER, B., LIU, J., KHESSIB, B., AND VAID, K. Ssd failures in datacenters: What? when? and why? In *Proceedings of the 9th ACM International on Systems and Storage Conference* (2016), SYSTOR '16, ACM, pp. 7:1–7:11.

[18] OPREA, A., AND JUELS, A. A clean-slate look at disk scrubbing. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2010), FAST'10, USENIX Association, pp. 5–5.

[19] PARIS, J.-F., SCHWARZ, T., AMER, A., AND LONG, D. D. E. Improving disk array reliability through expedited scrubbing. In *Proceedings of the 2010 IEEE Fifth International Conference on Networking, Architecture, and Storage* (Washington, DC, USA, 2010), NAS '10, IEEE Computer Society, pp. 119–125.

[20] 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* (2007), FAST '07, USENIX Association.

[21] SCHROEDER, B., DAMOURAS, S., AND GILL, P. Understanding latent sector errors and how to protect against them. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2010), FAST'10, USENIX Association.

[22] SCHROEDER, B., LAGISETTY, R., AND MERCHANT, A. Flash reliability in production: The expected and the unexpected. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies* (Berkeley, CA, USA, 2016), FAST'16, USENIX Association, pp. 67–80.

[23] WIKIPEDIA. S.m.a.r.t. https://en.wikipedia.org/wiki/S.M.A.R.T.

[24] ZHU, B., WANG, G., LIU, X., HU, D., LIN, S., AND MA, J. Proactive drive failure prediction for large scale storage systems. In *IEEE 29th Symposium on Mass Storage Systems and Technologies, MSST 2013, May 6-10, 2013, Long Beach, CA, USA* (2013), pp. 1–5.

Towards Production-Run Heisenbugs Reproduction on Commercial Hardware

Shiyou Huang
Texas A&M University
huangsy@tamu.edu

Bowen Cai
Texas A&M University
bowen.cai@tamu.edu

Jeff Huang
Texas A&M University
jeff@cse.tamu.edu

Abstract

We present a new technique, H3, for reproducing Heisenbugs in production runs on commercial hardware. H3 integrates the hardware control flow tracing capability provided in recent Intel processors with symbolic constraint analysis. Compared to a state-of-the-art solution, CLAP, this integration allows H3 to reproduce failures with much lower runtime overhead and much more compact trace. Moreover, it allows us to develop a highly effective core-based constraint reduction technique that significantly reduces the complexity of the generated symbolic constraints. H3 has been implemented for C/C++ and evaluated on both popular benchmarks and real-world applications. It reproduces real-world Heisenbugs with overhead ranging between 1.4%-23.4%, up to 8X more efficient than CLAP, and incurs only 4.9% runtime overhead on PARSEC benchmarks.

1 Introduction

The ability to reproduce software bugs is crucial for debugging, yet due to the often non-deterministic memory races among threads, it is notoriously difficult to reproduce concurrency bugs, *i.e.*, the so-called *Heisenbugs* [15]. Researchers have investigated significant efforts in *record & replay* (RnR) systems aiming to eliminate the non-determinism. However, it remains challenging to deploy an RnR system for production runs. Most existing solutions either are too slow due to the high runtime overhead incurred by tracing the shared memory dependencies, introduce the observer effect that makes the Heisenbugs disappear [17, 20, 31], or require special hardware that does not exist [16, 25, 26, 28, 33].

CLAP [18] introduces the idea of recording only thread-local information (*i.e.*, thread-local *control flow* paths) and then using offline constraint solving to reconstruct the shared memory dependencies. It is a promising solution for reproducing Heisenbugs because it does not record any cross-thread communication (data or synchronization); hence it requires no synchronizations during recording, which not only reduces the runtime overhead but also minimizes the observer effect.

To enable a production-run RnR solution, however, CLAP is still unsatisfactory due to two important challenges. First, although CLAP is much faster than conventional solutions, the runtime overhead incurred by CLAP using software path-recording is as large as 3X, which is unacceptable for most production environments. Second, the constraints generated by CLAP can be too complex to solve. In the worst case, the complexity of the constraints is exponential in the trace size. Despite that SMT solvers (e.g., Z3 [14]) are becoming increasingly powerful, in practice, the constraints can become too large to solve in a reasonable time.

In this paper, we present H3, a new RnR system to reproduce Heisenbugs by extending CLAP with commercial hardware features. Our key observation is that both of the aforementioned challenges can be effectively addressed by hardware-supported *control-flow* tracing. As also indicated in the CLAP paper [18], for path recording, hardware techniques [30] can achieve as low as 0.6% overhead. In reality, recent Intel processors (starting from the 5th generation) have provided a new feature called Processor Tracing (PT) to trace the program control flow with very small (less than 5%) runtime overhead [2]. PT uses highly-compacted packets (*i.e.*, only one bit for each conditional branch) to capture branch outcomes, often producing a compact trace requiring < 1 bit per retired assembly instruction. Moreover, hardware-supported tracing allows us to perform a significant reduction of the constraints generated by CLAP, because memory accesses executed on each core are ordered internally. We develop a core-based constraint reduction technique that reduces the complexity of the constraints from exponential in the trace size to only *exponential in the number of cores*.

As illustrated in Figure 1, H3 consists of two phases.

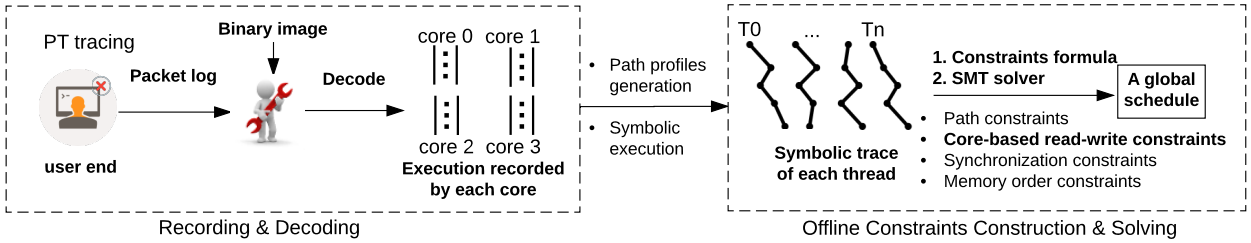


Figure 1: H3 Overview.

First, users run the target program on a COTS (commercial off-the-shelf) hardware with PT enabled. Once a failure occurs, the PT trace together with the thread context switch log are sent to the developer for reproducing the bug. From the PT trace and the binary image of the target program, H3 generates the instructions executed on each core. Second, H3 infers the instructions executed by each thread based on the thread context switch log and generates a symbolic trace for each thread. It then constructs symbolic constraints with the core-based constraint reduction, and computes a global failure reproducing schedule with an SMT solver.

Despite a clear design, realizing H3 faces two main additional technical challenges: 1) How to transform the low-level hardware trace to a high-level (source or IR) trace? 2) How to capture the data values (PT does not trace data values)? To solve the first challenge, we transform the hardware trace into a sequence of IR-level tuples, to identify what basic blocks are executed by each thread. This is done by matching the low-level assembly instructions in the per-thread local execution with that in the IR (*i.e.*, LLVM bitcode). For the second challenge, we symbolically execute the IR along the sequence of basic blocks for each thread. The unknown data values (including all the unknown read values and addresses) are encoded as symbolic variables, and are computed via constraint solving.

We implemented H3 for C/C++ programs based on PT, and evaluated it with a collection of popular performance benchmarks and real-world applications containing known Heisenbugs. Our experimental results show that H3 incurs only 1.4% to 23.4% runtime overhead for all the applications and only 4.9% for the PARSEC benchmarks on average, as much as 8X more efficient than CLAP. Moreover, H3 reduces the size of the constraints in CLAP by 28% to 99%, improving the speed of constraint solving by 2X-250X in most cases, and enabling H3 to reproduce more bugs than CLAP within a limited time budget.

This paper makes the following contributions:

- To our best knowledge, H3 is the first technique that integrates hardware control flow tracing with offline symbolic analysis for reproducing production-run Heisenbugs on commercial hardware.
- We develop a new core-based constraint reduction technique that significantly reduces the complexity of generated symbolic constraints from exponential in the trace size to exponential in the core counts.
- We implement and evaluate H3 on both popular benchmarks and real applications. Experiments show that H3 can reproduce real Heisenbugs in production runs with very small overhead.

2 Background

In this section, we first review the CLAP technique and elaborate its limitations. We then show how hardware control-flow tracing addresses these limitations.

2.1 CLAP

CLAP can not only reproduce Heisenbugs under sequential consistency (SC), but also a wide range of weak consistency memory models, including TSO (total store order) and PSO (partial store order) [9]. It has two key components: I) collecting per-thread control flow information via software path-recording (using an extended Ball-Larus path-recording algorithm [11]), and II) assembling a global schedule by solving symbolic constraints constructed over the thread local paths. To assemble a global schedule, CLAP has three steps:

1. Along the local path of each thread, it collects all the critical accesses (read, write or synchronization) to shared variables.
2. It introduces a fresh symbolic value for each read access, and collects the path constraints following the control flow for each thread via symbolic execution; it introduces an order variable for each critical access, and generates additional constraints according to synchronization, memory-consistency model, and potential inter-thread memory dependencies.
3. It uses an SMT solver to solve the constraints, to which the solutions correspond to global schedules that can reproduce the error. In other

words, the SMT solver computes what inter-thread memory dependencies would satisfy the memory-consistency model and enable the recorded local execution path.

CLAP contains several components to model a failing execution as constraints (*e.g.*, failure, path, synchronization, read-write, and memory model). We next use an example in Figure 2 to illustrate these constraints. Section 3.3 presents the constraint model in detail.

The program in Figure 2 contains a real Heisenbug that only manifests under the PSO memory model, which caused a $12 million financial loss in the real-world [7]. The root cause of the bug is that the write to z (line 5) can be reordered with the writes to x and y (lines 3-4) under PSO. The dashed arrow in the figure shows that the satisfaction of the *if* condition at line 7 depends on the write to z at line 5, which always happens after lines 3 and 4 under SC. However, under PSO, the write to z is allowed to happen before the write to y at line 4. As a result, when the *if* condition is satisfied, the value of $x+1$ and y may be unequal and hence triggering the error. The error can be triggered by the following PSO schedule: 1-2-3$_{R_x}$-3$_{W_x}$-4$_{R_y}$-5-7-8$_{R_x}$-8$_{R_y}$ (the subscripts are used to distinguish different operations from the same line).

The CLAP constraints for reproducing the buggy PSO schedule are shown in Figure 3. We use the order variable O_i denotes the order of the corresponding access at line i. The symbolic variable R_v^i denotes the value returned by the read access to the variable v at line i, and W_v^i the value written to v by the write at line i. To distinguish different operations at the same line, we add the type of the operation to the order variable. For example, $O_3^{R_x}$ and $O_3^{W_x}$ represent the orders of the read and write to x at line 3, respectively.

To manifest the error, CLAP enforces the assertion to be violated while satisfying the path constraints, i.e., $true \equiv (R_z^7 = 1 \wedge R_x^8 + 1 \neq R_y^8)$. A major part of the CLAP constraints is the read-write constraints, which are used to capture the potential inter-thread memory dependencies. Because the order of the memory accesses from different threads is unknown, the read-write constraints must encode a schedule for every potential read-write match, in which the read returns the value written by the write. For example, the read of z at line 7, R_z^7, may be matched with either the initial value 0, or the value written by line 2 or 5. If the former, the read R_z^7 should happen before all the writes to z; if the latter, R_z^7 should be matched with the corresponding write. For example, if R_z^7 returns the value by the write at line 2, the constraint $R_z^7 = W_z^2 \wedge O_2 < O_7 \wedge (O_5 < O_2 \vee O_7 < O_5)$ is generated.

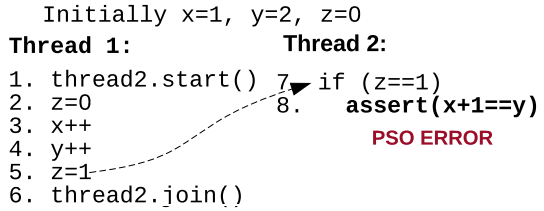


Figure 2: A real PSO bug in an electron microscope software [7], which caused a $12 million loss of equipment.

Read-Write Constraints

$$(R_z^7 = 0 \wedge O_7 < O_2) \vee (R_z^7 = W_z^2 \wedge O_2 < O_7 \wedge (O_5 < O_2 \vee O_7 < O_5)) \vee (R_z^7 = W_z^5 \wedge O_5 < O_7 \wedge (O_2 < O_5 \vee O_7 < O_2))$$

Memory Order Constraints

SC	PSO
$O_1 < O_2 < O_3^{R_x} < O_3^{W_x} < O_4^{R_x} < O_4^{W_x} < O_5 < O_6$ $O_7 < O_8^{R_x} < O_8^{R_y}$	$O_1 < O_2 \quad O_5 < O_6$ $O_3^{R_x} < O_3^{W_x} \quad O_4^{R_x} < O_4^{W_x}$ $O_7 < O_8^{R_x} < O_8^{R_y}$

Path Constraints	Failure Constraints
$R_z^7 = 1$	$R_x^8 + 1 ! = R_y^8$

Figure 3: The CLAP constraints for reproducing the PSO error in Figure 2. To save space, we show the read-write constraints for z only. Those for x and y are similar.

CLAP Limitations

1. Exponential complexity of read-write constraints. The read-write constraints generated by CLAP are very complicated in practice because there may exist many writes that a read can be matched with. In the worst case, the complexity of the read-write constraints (*i.e.*, the space of scheduling choices) is exponential in the number of writes (which typically accounts for a large percentage of the events in the trace). This is a bottleneck in CLAP especially for programs with intensive inter-thread memory dependencies, because the SMT solver may fail to solve the constraints. We will present a detailed complexity analysis in Section 3.4.

2. Slowdown of software path-recording. CLAP uses a highly optimized algorithm (i.e., Ball-Larus [11]) to track the control flow information for each thread. Although it greatly reduces the runtime overhead incurred by many other RnR solutions, it still incurs 10%-3X performance slowdown on popular benchmarks [18]. For instance, for the example in Figure 2, when the code is executed in a loop for 10 million times, CLAP incurs 2.3X program slowdown.

3. Difficulty of code instrumentation. It is difficult to apply software path-recording in production runs because it requires code instrumentation. Real-world programs often rely on external libraries, proprietary code, and/or are composed from layers of frameworks and extended by third-party plugins. Tracing the whole pro-

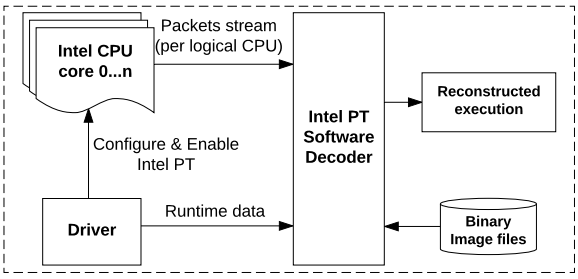


Figure 4: Components of Intel Processor Tracing (PT).

gram control flow by code instrumentation is difficult or impossible. For example, if a failure is caused by a bug in the uninstrumented external code, the constraints generated by CLAP may be incomplete and hence fail to reproduce the bug.

2.2 Hardware Control-Flow Tracing

Tracing control flow at the hardware level opens a door to apply CLAP in production runs by addressing the aforementioned limitations in three ways. First, hardware-supported control flow tracing is significantly more efficient than software-level path-recording. Compared to the 10%-3X overhead by software path-recording, PT achieves as low as 5% runtime overhead [2]. Second, hardware can track the full control flow of the code executed on each core. PT can not only trace the application code, but also the whole operating system kernel [2]. Third, tracing the control flow on each core enables a significant reduction of the complexity of the read-write constraints, because reads and writes from the same core are ordered already.

Next, we first review the basics of PT and then show its performance improvement over software path-recording on PARSEC 3.0 benchmarks [5].

Intel PT. As depicted in Figure 4, PT consists of two main components: tracing and decoding. For tracing, it only records the instructions that are related to the change of the program control flow and omits everything that can be deduced from the code (*e.g.*, unconditional direct jumps). For each conditional branch executed, PT generates a single bit (1/0) to indicate whether a conditional branch is taken or not taken. As such, PT tracks the control flow information, such as loops, conditional branches and function calls of the program, with minimal perturbation, and outputs a highly compact trace.

For decoding, PT provides a decoding library [1] to reconstruct the control flow from the recorded raw trace. It first synchronizes the packet streams with the synchronization packets generated during tracing, and then iterates over the instructions from the binary image to identify what instructions have been executed. Only when the decoder cannot decide the next instruction (e.g., when it encounters a branch), the raw trace is queried to guide the decoding process.

Table 1: Runtime and space overhead of PT on PARSEC.

Program	Native time (s)	PT time (s)	PT **OH(%)**	PT trace
bodytrack	0.557	0.573	**2.9%**	94M
x264	1.086	1.145	**5.4%**	88M
vips	1.431	1.642	**14.7%**	98M
blackscholes	1.51	1.56	**9.9%**	289M
ferret	1.699	1.769	**4.1%**	145M
swaptions	2.81	2.98	**6.0%**	897M
raytrace	3.818	4.036	**5.7%**	102M
facesim	5.048	5.145	**1.9%**	110M
fluidanimate	14.8	15.1	**1.4%**	1240M
freqmine	15.9	17.1	**7.5%**	2468M
Avg.	4.866	5.105	**4.9%**	553M

PT is configurable via a set of model-specific registers by the kernel driver. It provides a privilege-level filtering function for developers to decide what code to trace (*i.e.* kernel vs. user-space) and a CR3 filtering function to trace only a single application or process. PT on Intel Skylake processors also supports filtering by the instruction pointer (IP) addresses. This feature allows PT to selectively trace code that is only within a certain IP range, which can further reduce the tracing perturbation.

PT Performance. Table 1 reports the runtime and space overhead of PT on the PARSEC 3.0 benchmarks. We report the execution time of the programs without and with PT tracing (and the trace size), marked as native and PT respectively. Among the 10 benchmarks, PT incurs 1.4% to 14.7% runtime overhead (4.9% on average) and 88MB to 2.4GB space overhead (0.5GB on average).

3 H3

In this section, we present the technical details of H3. As we have described in Figure 1, H3 integrates hardware control-flow tracing with offline symbolic constraint analysis to reproduce Heisenbugs. Although the overall flow is easy to understand, there are three technical challenges in the integration:

1. **Absence of the thread information**. There is no thread information from the PT traces. It is unknown which instruction is executed by which thread, and hence difficult to construct the inter-thread synchronization and memory dependency constraints.

2. **Gap between low-level hardware traces and high-level symbolic traces**. The decoded execu-

tion from PT is in the low-level assembly form. However, to construct constraints and to reproduce bugs, we need a high-level symbolic trace containing shared variable accesses and branch conditions.

3. **No data values for memory accesses**. PT only traces control flow information but does not record any data values of memory accesses. To reconstruct the shared memory dependencies, we need a way to match reads with writes without using values.

We present our solutions to these challenges in the next three subsections. We also present a constraint reduction algorithm in Section 3.4 enabled by the partial order of writes per-core, which significantly reduces the complexity of the generated constraints.

3.1 Thread Local Execution Generation

We leverage the context-switch software events (generated by the Linux Perf tool) to distinguish instructions from different threads. Each context-switch event contains three attributes: TID, CPUID, and TIME (*i.e.*, the timestamp of the event). Because PT also generates frequent synchronization packets (including the timestamp information) into the packet stream, we can use the timestamp information to synchronize the context switch events with the PT packets from the same core (*i.e.*, CPUID). Because the timestamp clocks local to each core is precise, the inferred thread identity based on the timestamp information is also precise. Hence, we locate the context switch points in the PT packets on each core by comparing the timestamps, and determine the thread identity of each instruction as the TID attribute of the leading context-switch event.

3.2 Symbolic Trace Generation

In CLAP, the symbolic trace of each thread is generated by symbolic execution along the recorded path profile of each thread. The path profile for each thread is decoded (from the Ball-Larus path encoding [11]) as a sequence of basic block transitions at the LLVM IR level in the form of (*Tid, BasicBlockId*). In H3, we also rely on these high-level per-thread path profiles to collect the symbolic traces, and we extract the path profiles from the low-level PT trace as follows. We first instrument all basic blocks of the target program and assign each a unique identifier. Then we compare the generated assembly code from the instrumented program with the decoded instructions from the PT trace to identify which basic blocks are executed by each thread.

Algorithm 1 shows the process of generating the path profiles for each thread. The algorithm takes as input: (1) the executed instructions and their corresponding line number; and (2) the basic blocks of the control-flow of the program with the *BlockId* and the line number of the first instruction of this block. The algorithm first gets the executed instructions by each thread (line 3) and then matches the line number of the executed instructions with that contained in each basic block (line 4-7). To identify the path profile of a thread, the algorithm iterates over the instructions of each thread to check whether the instruction is the first one of the block by comparing the line number (line 5). If so, we add this block into the path profile as (*Tid, BasicBlockId*).

Algorithm 1 Path profiles generation

Input: $\mathbb{L}$:< *line, insn* > //execute instructions and #line
Input: $\mathbb{B}$: <*line, block_id*> //basic blocks of the paths
Output: $\mathbb{Q}$: <*tid, block_id*>//path profile of each thread

```
1: for each tid do    //traverse each thread
2:     //get the instructions of each thread
3:     ℓ = {𝕊 ⊆ 𝕃 | ∀insn ∈ 𝕊.insn, Tid(insn) = tid}
4:     for each item ∈ ℓ do
5:         if item.line ∈ 𝔹.line then
6:             block_id = 𝔹.get(item.line)
7:             ℚ.add(tid, block_id)
8: return ℚ
```

3.3 Matching Reads and Writes

To reconstruct the shared memory dependencies without data values, similar to CLAP, we construct a system of symbolic constraints over the per-thread symbolic traces. The basic idea is to introduce an order variable for each read/write denoting the unknown scheduling order, and a symbolic variable for each read/address denoting the unknown read value and address. We symbolically execute the program following the recorded per-thread control flow, and constructs constraints over the order and symbolic variables to determine the inter-thread orders and values of reads/addresses.

More specifically, we construct a system of SMT constraints formula, denoted by Φ_g, over the symbolic traces. The computed orders/values from solving Φ_g then correspond to one or more concrete global schedules that can reproduce the Heisenbugs. We note that the computed schedules may be different from that in the failure execution, but any one of them is sufficient to reproduce the Heisenbugs.

Φ_g can be decomposed into five parts:

$$\Phi_g = \Phi_{path} \wedge \Phi_{bug} \wedge \Phi_{sync} \wedge \Phi_{mo} \wedge \Phi_{rw}$$

where Φ_{path} denotes the path conditions by each thread; Φ_{bug} the condition for the bug manifestation; Φ_{sync} the interactions between inter-thread synchronizations; Φ_{rw}

the potential inter-thread memory dependencies; and Φ_{mo} the memory model constraints. The formula contains two types of variables: (1) *V - the symbolic value variables* denoting the values returned by reads; and (2) *O - the order variables* the order of each operation in the final global schedule.

Path Constraints (Φ_{path}). The path constraints are constructed by a conjunction of all the path conditions of each thread, with each path condition corresponds to a branch decision by that path. The path conditions are collected by recording the decision of each branch via symbolic execution.

Bug Constraints (Φ_{bug}). The bug constraints enforce the conditions for a bug to happen. A bug can be a crash segfault, an assert violation, a buffer overflow, or any program state-based property. To construct the bug constraints, an expression over the symbol values for satisfying the bug conditions is generated. For example, the violation of an assertion *exp* can be modeled as !*exp*.

Synchronization Constraints (Φ_{sync}). The synchronization constraints consist of two parts: partial order constraints and locking constraints. The partial order constraints model the order between different threads caused by synchronizations *fork/join/signal/wait*. For example, The *begin* event of a thread t should happen after the *fork* event that starts t. A *join* event for a thread t should happen after the last event of t. The locking constraints ensures that events guarded by the same lock are mutually exclusive. It is constructed over the ordering of the *lock* and *unlock* events. More specifically, for each lock, all the *lock/unlock* pairs of events are extracted, and the following constraints for each two pairs (l_1, u_1) and (l_2, u_2) are constructed: $O_{u_1} < O_{l_2} \vee O_{u_2} < O_{l_1}$.

Memory Order Constraints (Φ_{mo}). The memory order constraints enforce orders specified by the underlying memory models. H3 currently supports three memory models: SC, TSO and PSO. For SC, all the events by a single thread should happen in the program order. TSO allows a read to complete before an earlier write to a different memory location, but maintains a total order over writes and operations accessing the same memory location. PSO is similar to TSO, except that it allows re-ordering writes on different memory locations.

Read-Write Constraints (Φ_{rw}). Φ_{rw} matches reads and writes by encoding constraints to enforce the read to return the value written by the write. Consider a read r on a variable v and r is matched to a write w on the same variable; we must construct the following constraints: the order variables of all the other writes that r can be matched to are either less than O_w or greater than O_r.

As discussed in Section 2.1, Φ_{rw} can be complicated because there may exist many potential matches between reads and writes. The size of Φ_{rw} is cubic in the trace size and its complexity is exponential in the trace size. Nevertheless, in next subsection, we show that both the size and complexity of Φ_{rw} can be greatly reduced in H3.

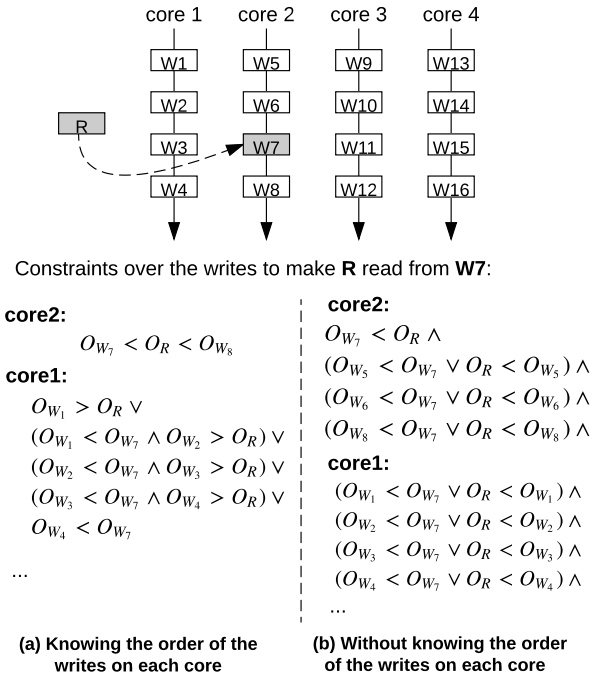


Figure 5: Core-based constraint reduction.

3.4 Core-based Constraints Reduction

Besides the low runtime overhead, another key innovation enabled by PT is that the order of executed events on each core (either by the same thread or by different threads) is determined, which can reduce the complexity of Φ_{rw} from exponential in the number of writes to exponential in the core counts.

The key observation of this reduction is that the executed memory accesses on each core decoded from PT trace are already ordered, following the program order. Once the order of a certain write in the global schedule is determined, all the writes that happen before or after this write, on the same core, should occur before or after this write in the schedule correspondingly. This eliminates a large number of otherwise necessary read-write constraints for capturing the potential inter-thread memory dependencies.

Consider an example in Figure 5, which has four cores with each executing four different writes. Suppose there is a read R that can be potentially matched with all of these writes, because each of them writes a different value to the same shared variable read by R. Without the partial order information of each core, we must include all writes and their orderings into the constraints.

For instance, if R reads the value from the write W_7 on Core 2, then R must happen after W_7 (*i.e.*, $O_R > O_{W_7}$), and all the other writes must either happen before W_7 or after R. Taking W_5 as an example; it must either happen before W_7 or after the read R, resulting in the constraint $(O_R < O_{W_5} \vee O_{W_5} < O_{W_7})$. In general, if there are N writes in the trace, the constraints can generate 2^N different ordering choices for these writes. As typically most accesses in the trace are reads and writes, this exponential search space can be a bottleneck for the technique to scale.

However, with the per-core partial order information, the execution order of the writes on each core is already determined. To prevent other writes from happening between the considered write and read, we only need to take the read-write as a whole and insert it to those sorted writes. Algorithm 2 presents our constraints reduction algorithm. Following this algorithm, to make R read from W_7, for all the other writes on Core 2, we only require $O_{W_7} < O_R < O_{W_8}$. Moreover, for the writes on the other cores, our new constraints encode fewer ordering choices. For example, for the four writes (W_1-W_4) on Core 1, the constraints are written as $O_R < O_{W_1} \vee (O_{W_1} < O_{W_7} \wedge O_R < O_{W_2}) \vee (O_{W_2} < O_{W_7} \wedge O_R < O_{W_3}) \vee (O_{W_3} < O_{W_7} \wedge O_R < O_{W_4}) \vee O_{W_4} < O_{W_7})$. There are only 5 ordering choices (compared to 16 in CLAP).

Algorithm 2 Core-based constraints reduction

Input: a matched read-write $< R, W >$
Output: Φ_{rw} to make R read from W
1: *Initial*: $\Phi_{rw} = \emptyset$
2: ***case 1***: *writes executed on the same core as W*
3: $\Phi_{rw} = \Phi_{rw} \wedge (O_W < O_R < O_{W'})$ //W' happens right after W on the same core
4: ***case 2***: *writes executed on other cores*
5: //for any two writes W_i and W_{i+1} on the same core
6: $\Phi_{rw} = \Phi_{rw} \wedge (O_R < O_{W_i} \vee (O_{W_i} < O_W \wedge O_R < O_{W_{i+1}})) \vee O_W > O_{W_{i+1}}$
7: *return* Φ_{rw}

We note that the core-based constraints apply to SC and TSO, but may not apply to those weak memory models that allow re-ordering of writes on the same core. The reason is that if writes are re-ordered, the partial order witnessed on each core may not reflect the actual buggy execution order.

Theorem 1 below states the soundness guarantee of the core-based reduction:

Theorem 1 *If a concurrent program runs on an SC or TSO platform with C cores and there are N writes executed, the number of the ordering choices of the read-write constraints is reduced from* 2^N *to* $(\frac{N}{C}+1)^C$.

Proof. Consider that a read R returns the value of a write W. When not knowing the partial order of the writes on each core, each write either happens before W or after R. Consequently, there are 2^N ordering choices in total. If the partial order of the writes on each core is known and each core contains $m_i = \frac{N}{C}$ writes, the ordering on each core has only $m_i + 1$ choices. Therefore, the total number of choices is reduced to $\Pi_{i=1}^{C}(m_i+1)$, which equals to $(\frac{N}{C}+1)^C$.

4 Implementation

We have implemented H3 for Pthreads-based C/C++ programs based on a number of tools, including CLAP [18], the Linux Perf Tools [3], the PT decoding library [1], and the Z3 SMT solver [14]. We use Perf to control Intel PT to collect the packet streams and the context switch events. We first insert the context switch events to the packet streams by comparing the timestamp information, and then use the PT decoding library to decode the packets information. As in CLAP, we use KLEE [12] as the symbolic execution engine to generate the symbolic traces for each thread, and construct an SMT constraint formula. We modified CLAP to implement the core-based constraint reduction algorithm, and we use Z3 to solve the constraints.

Shared Variable Identification. We first run a static thread sharing analysis based on the Locksmith [29] race detector and then manually mark each shared variable `x` as symbolic by `klee_make_symbolic(&x, sizeof(x), "x")`, like CLAP. One way to automate this step is to conservatively consider all variables in the program as potentially shared and marked them as symbolic. However, this would produce a large amount of unnecessary constraints. For external function calls that are not supported by KLEE, we also mark the input and return variables of the external function calls as symbolic.

Constraint Reduction. For the core-based constraint reduction, we first extract the writes on the same core from the PT trace and store these writes in a map (*core Id:* $w_1[line], w_2[line]...$). When constructing the read-write constraints, this map is used to determine which write belongs to which core by comparing the associated line number information. Because all writes on the same core occur in the order that they are executed, we construct a happens-before constraint over these writes. When matching a read r to a corresponding write w, we first constrain r to happen after w and happen before the write that occurs right after w on the same core, and we

Table 2: Benchmarks.

Program	LOC	#Threads	#SV	#insns (executed)	#branches (total)	#branches (app)	Ratio app/total	Symb. time
racey	192	4	3	1,229,632	78,117	77,994	99.8%	107s
pfscan	1026	3	13	1,287	237	43	18.1%	2.5s
aget-0.4.1	942	4	30	3,748	313	5	1.6%	117s
pbzip2-0.9.4	1942	5	18	1,844,445	272,453	5	0.0018%	8.7s
bbuf	371	5	11	1,235	257	3	1.2%	5.5s
sbuf	151	2	5	64,993	11,170	290	2.6%	1.6s
httpd-2.2.9	643K	10	22	366,665	63,653	12,916	20.3%	712s
httpd-2.0.48	643K	10	22	366,379	63,809	13,074	20.5%	698s
httpd-2.0.46	643K	10	22	366,271	63,794	12,874	20.2%	643s

then only need to disjunct the order constraints between w and those writes from a different core.

5 Evaluation

Our evaluation of H3 focuses on answering two sets of questions:

- How is the runtime performance of H3? How much runtime improvement is achieved by H3 compared to CLAP?
- How effective is H3 for reproducing real-world Heisenbugs? How effective is the core-based constraint reduction technique?

5.1 Methodology

We evaluated H3 with a variety of multithreaded C/C++ programs collected from previous studies [18, 35, 6], including nine popular real-world applications containing known Heisenbugs. Table 2 summarizes these benchmarks. *pfscan* is a parallel file scanner containing a known bug; *aget-0.4.1* is a parallel *ftp/http* downloading tool containing a deadlock; *pbzip2-0.9.4* is a multithreaded implementation of *bzip* with a known order violation; *bbuf* is shared bounded buffer and *sbuf* is a C++ implementation of the JDK1.4 *StringBuffer* class; *httpd-2.2.9*, *httpd-2.0.48*, *httpd-2.0.46* are from the Apache HTTP Server each containing a known concurrency bug; We also included *racey* [6], a special benchmark with intensive races that are designed for evaluating RnR systems. We use *Apache Bench* (*ab*) to test *httpd*, which is set to handle 100 requests with a maximum of 10 requests running concurrently.

We compared the runtime performance of H3 and CLAP by measuring the time and space overhead caused by PT tracing and software path-recording. We ran each benchmark five times and calculated the average. All experiments were performed on a 4 core 3.5GHz Intel i7 6700HQ Skylake CPU with 16 GB RAM running Ubuntu 14.04.

We evaluated the effectiveness of H3 for reproducing bugs by checking if H3 can generate a failure reproducing schedule and by measuring the time taken by offline constraint solving. We set one hour timeout for Z3 to solve the constraints.

For most benchmarks, the failures are difficult to manifest because the erroneous schedule for triggering the Heisenbugs is rare. Similar to CLAP, we inserted timing delays (*sleep* functions) at key places in each benchmark and executed it repeatedly until the failure is produced. We also added the corresponding assertion to denote the bug manifestation.

Benchmark Characteristics. Table 2 reports the execution characteristics of the benchmarks. Columns 3 and 4 report the number of threads and shared variables, respectively, contained in the execution. We also profiled the total number of the executed instructions and branches in the assembly code, and the branches from the LLVM IR code, as reported in Columns 5-7. Column 8 reports the ratio of the number of the branches in the instrumented application code versus the total number of branches (in both the application code and all the external libraries). For most benchmarks (except *racey*), the ratio is smaller than or around 20%. Column 9 reports the time for constructing the symbolic trace for the corresponding recorded execution of the benchmark.

5.2 Runtime Performance

Table 3 reports the performance comparison between H3 and CLAP. Column 2 reports the native execution time of the benchmarks. Columns 3-4 report the execution time with H3 and CLAP and their runtime overhead. Column 5 reports the speedup of H3 over CLAP. Column 6 reports the percentage of branch instructions in the execution. This number is proportional to the runtime

Table 3: Performance comparison between H3 and CLAP.

Program	Native time (s)	Time (s)			Branch insts%	Space overhead	
		CLAP (**Overhead**)	H3 (**Overhead**)	Speedup		CLAP	H3
racey	0.268	0.768(186.6%)	0.288(**7.5%**)	**65.2%**	6.4%	96M	2.68M
pfscan	0.094	0.104(11.0%)	0.116(**23.4%**)	-11.5%	18.4%	3.2K	30K
aget-0.4.1	0.139	0.156 (12.1%)	0.152(**9.4%**)	**2.6%**	17.9%	11K	41K
pbzip2-0.9.4	0.102	0.134(31.4%)	0.112(**9.8%**)	**16.4%**	14.8%	5.2K	677K
bbuf	0.232	0.696(200%)	0.264(**13.8%**)	**62.1%**	20.1%	3.9K	2.7M
sbuf	0.216	0.299(38.5%)	0.256(**18.5%**)	**14.4%**	17.2%	6.6K	4.5M
httpd-2.2.9	0.53	0.71(34.0%)	0.57(**7.5%**)	**19.7%**	17.4%	7.8M	10.43M
httpd-2.0.48	0.45	0.59(32.1%)	0.51(**13.3%**)	**13.6%**	17.4%	8.1M	11.79M
httpd-2.0.46	0.42	0.57(36.2%)	0.50(**19.0%**)	**12.3%**	17.4%	7.2M	10.62M
avg.	0.272	0.447(64.3%)	0.307(**12.9%**)	**31.3%**	16.3%	13.2M	4.8M

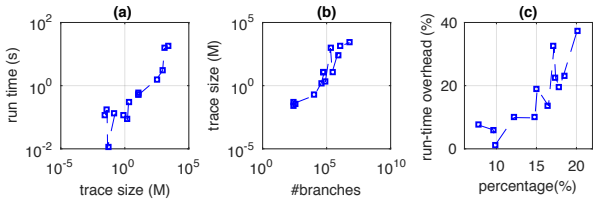


Figure 6: H3 performance analysis.

overhead of PT. Columns 7-8 report the space overhead of H3 and CLAP, respectively.

Overall, the runtime overhead of H3 on these benchmarks ranges between 7.5%-23.4% and 12.9% on average. Compared to CLAP (11.0%-2X overhead), H3 achieves as much as 8X performance improvement and reduces its overhead significantly by 2.6%-65.2% and 31.3% on average. The only exception is *pfscan*. However, this is just because *pfscan* contains significantly more external calls compared to the other benchmarks; while H3 records all external library calls, the implementation of CLAP does not (which sacrifices the correctness). In addition, the short execution time of *pfscan* can suffer from noise.

For space overhead, H3 produces 30KB-2.4GB traces on these benchmarks, whereas CLAP produces 2KB-2.1GB. Some numbers of CLAP are smaller than that of H3, because external library calls are not traced by CLAP.

H3 performance analysis. We note that the performance of H3 is dominated by PT for tracking the control flow events. The additional cost for H3 to track context switching events is almost negligible as compared to tracing the control flow. We have also evaluated the runtime performance of H3 on the PARSEC 3.0 benchmarks and found that H3 incurs only 1.4% to 14.7% runtime overhead (4.9% on average) and 0.5GB trace size, the same as that reported in Table 1 for PT.

We further conducted a performance study of H3 on PARSEC with respect to three impacting factors: the trace size, the number and percentage of branch instructions, as shown in Figure 6. Figure 6(a) shows the relation between the size of the recorded trace and the execution time of H3. Figure 6(b) shows the relation between the number of executed branches and the size of the recorded trace. Figure 6(c) shows that relation between the percentage of executed branch instructions and the runtime overhead of H3. The results indicate that the performance of H3 is proportional to the percentage of executed branch instructions in the execution. Recall Column 8 in Table 2 that the number of branches in the application code often accounts for a small percentage of the total number of branches. Hence, in practice, the performance of H3 can be further improved by tracing only the application code and omitting external library calls.

5.3 Effectiveness of Bug Reproduction

Table 4 reports the results of Heisenbug reproduction. We successfully evaluated five benchmarks[1] with a total number of seven Heisenbugs. *racey1*, *racey2* and *racey3* correspond to the *racey* benchmark with 500, 1000, and 1500 loop iterations.

Column 2 reports the number of unknown variables in the constraint formula, corresponding to the number of read/write/synchronization operations in the symbolic trace. Columns 3-6 report the results of CLAP, including the total size of the generated constraints (in terms of the number of constraint clauses), the size of read-write constraints, the constraint solving time by Z3 and whether Z3 returns a solution before timeout in one hour. Columns 7-10 report the corresponding results of H3.

Overall, H3 is more efficient and effective than CLAP in reproducing Heisenbugs. The key difference between H3 and CLAP is that with the core-based constraint reduction, H3 generates a much simpler and smaller con-

[1]We excluded *aget* and the *httpd* benchmarks because the KLEE symbolic execution failed on them.

Table 4: Results of Heisenbug reproduction. (-) means the solver runs timeout in one hour.

Program	#Var	CLAP #constraints		solve time	success?	H3 #constraints		solve time	success?
		#Total	#RW			#Total	#RW(**Reduction**)		
bbuf	79	14264	13902	98s	Y	10344	9982(**28.2%**)	52s	Y
sbuf	102	438	302	1s	Y	344	208(**31.1%**)	1s	Y
pfscan	25	199	60	1s	Y	179	40(**33.3%**)	1s	Y
pbzip2	113	5890	1270	2s	Y	5460	840(**33.9%**)	1s	Y
racey1	15040	540602	540388	-	N	50602	50388(**90.7%**)	267s	Y
racey2	30108	41612000	41607900	-	N	201202	200788(**99.5%**)	-	N
racey3	67850	1.3×10^8	1.3×10^8	-	N	451802	451188(**99.7%**)	-	N

straint formula than CLAP. H3 reduces the size of the CLAP constraints by 28%-99%, and is able to reproduce more bugs than CLAP. Both H3 and CLAP reproduce the bugs in the four benchmarks *bbuf*, *sbuf*, *pfscan* and *pbzip2*. H3 additionally reproduces the bug in *racey1*, while CLAP fails because the solver could not solve the constraints in time. In addition, for *bbuf*, although both H3 and CLAP can reproduce the bug, H3 is much faster (52s vs 98s) than CLAP. H3 fails on *racey2* and *racey3* because the constraints in these two cases are still too complex to solve.

6 Limitations and Future Work

Our experimental results show that H3 achieves a significant performance improvement over CLAP by integrating hardware control-flow tracing with constraint analysis. Nevertheless, we observe several factors that can be leveraged to further improve the performance of H3.

Large PT Trace Data. On our current platform, the size of the PT trace buffer per core is limited to 4MB. For tracing long running programs, the buffer can get full quickly (e.g., 0.01s for the PARSEC benchmarks). Currently, Perf actively monitors the trace buffer and flushes it to disk once the buffer is full. To avoid overwriting the buffered data, Perf also needs to disable PT when the buffer is full, and wakes it up when the data is copied out. This is a main bottleneck that limits the runtime performance of H3 because the program execution has to be suspended when PT is off, otherwise the control flow data may be lost when the buffer data is being copied out. We also experienced data loss with Perf when using PT to track long traces. This happens because the speed of copying data out is not fast enough, causing certain buffered data overwritten by the new data. We expect that a larger trace buffer or double buffering in the future generations of PT will help alleviate this problem.

Data Values. Another limitation of PT is that it only tracks the control flow of the program but not any data values or memory addresses. This is the main reason why symbolic execution is needed in H3 to construct symbolic traces. Although symbolic execution engines such as KLEE are becoming increasing powerful, scaling symbolic execution to long running programs remains a challenging problem. In addition, limited by KLEE, H3 currently can only reproduce concurrency failures that occur in the application code, but not external function calls (though it traces the control flow in all external libraries).

For future work, we plan to use hardware watchpoints (as also used in Gist [19]) to capture the value and address of variables along with the PT control flow tracing. With the value information, we can then skip the symbolic execution part but construct the constraints by matching the values of reads and writes directly. Moreover, this will further reduce the complexity of the generated constraints.

Constraint Solving for Long Traces. Although our constraint reduction is effective, the complexity of the generated constraints is still exponential in the number of cores. For long traces, the constraint size can still be large and solving them remains challenging. For example, H3 failed on *racey2* and *racey3* due to the solver timeout. For this problem, we plan to improve H3 in two ways. First, we can perform periodic checkpoints (e.g., using the snapshot mode of Perf) to save the current state of the program, such that when a failure occurs, H3 needs only to generate the constraints from the last checkpoint to the failure. Second, we can reduce the amount of the trace by not tracing the control flow in the external libraries (e.g., using the IP filtering featured supported by Skylake processors). As shown in our experimental results, the branches from the application code account for only a small percentage (7-20%) of the total trace, most of which are from the external libraries. Skipping tracing the external libraries will greatly reduce both the trace size and the runtime overhead.

Non-deterministic Program Inputs. Similar to CLAP, currently H3 does not record the program input but assumes that all program inputs are fixed. If the program input is non-deterministic or certain program inputs are missed, H3 may fail to reproduce the bug. This prob-

lem can be addressed by tracking the program input and enforcing the same input value during the symbolic trace construction and the bug reproduction. Mozilla RR [4] is a promising solution to track non-deterministic inputs in real-world systems, by tracing only system call results and signals with `ptrace`. We expect that by integrating H3 with RR, H3 will be able to reproduce failures resulted from both non-deterministic schedules and inputs.

7 Related Work

Researchers have proposed many different RnR systems, both at the software level [8, 13, 17, 18, 21, 22, 23, 27, 32, 34, 38] and hardware-level [16, 25, 26, 28, 33]. Most RnR systems are either order-based [13, 17, 23, 34, 38] that rely on faithfully recording the shared memory dependencies at runtime, or search-based [8, 18, 21, 22, 32] that record only partial information at runtime and rely on powerful search engines such as SMT solvers to reconstruct the memory dependencies.

A central goal of RnR systems is to reduce the runtime overhead such that they can be used in production runs. Hardware techniques [16, 25, 26, 28, 33] are often much more efficient than software-level implementation, but most previous RnR systems rely on special hardware that is not available. Intel PT is an exciting hardware feature that opens a door for RnR systems to be applied broadly in COTS platforms.

Gist [19] introduces a bug diagnosis technique that also leverages PT to identify root causes of a failure with low overhead. Different from H3, Gist assumes the failure can be reproduced in the first place, but it may fail to do so. In addition, Gist relies on statistical analysis to identify failure causes, but it has no guarantee, i.e., it may miss real causes or report false positives. Compared to Gist, H3 solves a different problem: reproducing failures before they can be diagnosed, and H3 is sound: it guarantees to reproduce the failure as long as the constraints can be solved by the solver.

Arulraj *et al.* [10] use hardware performance counters for failure diagnosis. This technique leverages the hardware to sample predicates from a large number of successful and failing runs and then use the sampled predicates to diagnose the failure via statistical analysis.

ReCBuLC [36] uses hardware clocks that are available on modern processors to help reproducing Heisenbugs. The recorded timestamps local to each thread together with a statistical analysis for calculating the time differences among local clocks across different cores, are used to determine the global schedule of shared-resource accesses. One limitation of this approach is that the statistical analysis may fail to infer a correct global schedule.

The idea of using offline constraint analysis to infer global failure schedules was pioneered by Lee *et al.* [21, 22]. The technique uses load-based checkpoints to search for a global schedule without recording any shared memory dependencies. However, compared to PT, the load-based checkpoints are not supported by the commodity architecture.

Similar to CLAP, both ODR [8] and Symbiosis [24] rely on symbolic constraint solving to figure out schedules that can satisfy certain conditions. ODR uses constraints to reproduce failures, and Symbiosis uses constraints for reducing the schedule complexity.

PRES [27] proposes a probabilistic replay technique that uses an intelligent feedback-based replayer to reproduce failures with lightweight recording. PRES may fail to reproduce the bug in the first attempt due to a recorded incomplete schedule. However, it can learn from the previous failing replays to rectify the schedule. Typically after a few attempts, PRES is able to find a correct schedule to reproduce the bug.

Both CoreDump [32] and ESD [37] rely on only the program coredumps to diagnose failures. CoreDump uses a technique called execution indexing to compare the differences between coredumps from failing and normal runs to identify the failing point. ESD uses static analysis and symbolic execution to synthesize both program inputs and schedule to reproduce failures. Using coredumps is promising for diagnosing real-world failures since coredumps are often available after the program crash. However, since there is no program control flow information, the technique may be difficult to reproduce failures that require complex paths and schedules to manifest.

8 Conclusion

We have presented H3, a novel technique that reproduces Heisenbugs by integrating hardware control flow tracing and symbolic constraint solving. With the efficient control flow tracing supported by PT, H3 enables for the first time the ability to efficiently reproduce Heisenbugs in production runs on commercial hardware. We have also presented an effective core-based constraint reduction technique that significantly reduces the size of the symbolic constraints and hence scales H3 to larger programs compared to the state-of-the-art solutions. Our evaluation on both popular benchmarks and real-world applications shows that H3 can effectively reproduce Heisenbugs in production runs with very small overhead, 4.9% on average on PARSEC.

Acknowledgement

We would like to thank our shepherd, Gilles Muller, and the anonymous reviewers for their valuable feedback. This work was supported by NSF award CCF-1552935.

References

[1] Intel processor trace decoder library. https://github.com/01org/processor-trace.

[2] Intel PT Micro Tutorial. https://sites.google.com/site/intelptmicrotutorial.

[3] Linux perf documentation. https://github.com/torvalds/linux/tree/master/tools/perf.

[4] Mozilla rr. https://github.com/mozilla/rr.

[5] The PARSEC benchmarks. http://parsec.cs.princeton.edu/.

[6] Racey: A stress test for deterministic execution. http://pages.cs.wisc.edu/~markhill/racey.html.

[7] A real-world bug caused by relaxed consistency. http://stackoverflow.com/questions/16159203.

[8] G. Altekar and I. Stoica. Odr: Output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP, 2009.

[9] T. S. architecture manual. Version 9. *SPARC International, Inc.* 1994.

[10] J. Arulraj, P.-C. Chang, G. Jin, and S. Lu. Production-run software failure diagnosis via hardware performance counters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2013.

[11] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, 1996.

[12] 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 USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2008.

[13] Y. Chen and H. Chen. Scalable deterministic replay in a parallel full-system emulator. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, 2013.

[14] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[15] J. Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12. IEEE Computer Society, 1986.

[16] D. R. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA, 2008.

[17] J. Huang, P. Liu, and C. Zhang. Leap: Lightweight deterministic multi-processor replay of concurrent java programs. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE, 2010.

[18] J. Huang, C. Zhang, and J. Dolby. Clap: Recording local executions to reproduce concurrency failures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2013.

[19] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP, 2015.

[20] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy. Chimera: Hybrid program analysis for determinism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2012.

[21] D. Lee, M. Said, S. Narayanasamy, and Z. Yang. Offline symbolic analysis to infer total store order. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 357–358. IEEE, 2011.

[22] D. Lee, M. Said, S. Narayanasamy, Z. Yang, and C. Pereira. Offline symbolic analysis for multi-processor execution replay. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, 2009.

[23] P. Liu, X. Zhang, O. Tripp, and Y. Zheng. Light: Replay via tightly bounded recording. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2015.

[24] N. Machado, D. Quinta, B. Lucia, and L. Rodrigues. Concurrency debugging with differential schedule projections. *ACM Trans. Softw. Eng. Methodol.*, 25(2), Apr. 2016.

[25] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution ef?ciently. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA, 2008.

[26] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA, 2005.

[27] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. Pres: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP, 2009.

[28] G. Pokam, C. Pereira, S. Hu, A.-R. Adl-Tabatabai, J. Gottschlich, J. Ha, and Y. Wu. Coreracer: A practical memory race recorder for multicore x86 tso processors. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, 2011.

[29] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Context-sensitive correlation analysis for race detection. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, 2006.

[30] K. Vaswani, M. J. Thazhuthaveetil, and Y. N. Srikant. A programmable hardware path profiler. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO, 2005.

[31] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2011.

[32] D. Weeratunge, X. Zhang, and S. Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2010.

[33] M. Xu, R. Bodik, and M. D. Hill. A" flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 122–133. IEEE, 2003.

[34] Z. Yang, M. Yang, L. Xu, H. Chen, and B. Zang. Order: Object centric deterministic replay for java. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC, 2011.

[35] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 485–502, 2012.

[36] X. Yuan, C. Wu, Z. Wang, J. Li, P.-C. Yew, J. Huang, X. Feng, Y. Lan, Y. Chen, and Y. Guan. Recbulc: Reproducing concurrency bugs using local clocks. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE, 2015.

[37] C. Zamfir and G. Candea. Execution synthesis: A technique for automated software debugging. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys, 2010.

[38] J. Zhou, X. Xiao, and C. Zhang. Stride: Search-based deterministic replay in polynomial time via bounded linkage. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE, 2012.

A DSL Approach to Reconcile Equivalent Divergent Program Executions

Luís Pina Daniel Grumberg Anastasios Andronidis Cristian Cadar
Department of Computing
Imperial College London, UK
{*l.pina, daniel.grumberg14, a.andronidis15, c.cadar*}*@imperial.ac.uk*

Abstract

Multi-Version Execution (MVE) deploys multiple versions of the same program, typically synchronizing their execution at the level of system calls. By default, MVE requires all deployed versions to issue the same sequence of system calls, which limits the types of versions which can be deployed.

In this paper, we propose a Domain-Specific Language (DSL) to reconcile expected divergences between different program versions deployed through MVE. We evaluate the DSL by adding it to an existing MVE system (Varan) and testing it via three scenarios: (1) deploying the same program under different configurations, (2) deploying different releases of the same program, and (3) deploying dynamic analyses in parallel with the native execution. We also present an algorithm to automatically extract DSL rules from pairs of system call traces. Our results show that each scenario requires a small number of simple rules (at most 14 rules in each case) and that writing DSL rules can be partially automated.

1 Introduction

Multi-version execution (MVE) has seen a revival in recent years as a mechanism to increase software security and reliability [13, 18, 20, 22, 29, 34, 35]. At a high-level, MVE works by running multiple versions of a program in parallel, synchronizing their execution typically at the level of system calls. In a security context, one can run diversified program variants (e.g., where each variant has a different memory layout) in such a way that divergences across variants signal a security attack [29, 34]. In a reliability context, one can run diversified variants or multiple software revisions and allow the overall application to continue execution when versions crash [18, 19].

In its initial instantiation, MVE employs a monitor process that intercepts all the system calls issued by the underlying versions. When all versions issue the same system call, the monitor executes the system call once on behalf of all versions, and copies the results to each version. If any version diverges, i.e. issues a different system call, the monitor raises a warning and stops executing (in a security context) or terminates the divergent versions and MVE continues with fewer versions (in a reliability context).

There are two main issues with this simple form of MVE. First, executing system calls from all versions in lock-step imposes a large performance penalty. Second, this form of MVE relies on all versions issuing the *same* sequence of system calls. The latter issue is particularly problematic because it limits the types of versions that can be run with MVE. For instance, the diversified variants cannot issue different but equivalent sequences of system calls (e.g., those arising due to refactoring), and the MVE system cannot ignore additional system calls (e.g., that one version may use for extra logging).

A new architecture, recently introduced by Varan [19], tackles both issues. In the proposed scheme, which resembles an in-memory record-replay framework, there is no central monitor. Instead, one of the versions acts as the *leader* and executes system calls directly, writing their results into a shared ring buffer. The other versions, *followers*, simply read back the results from the ring buffer (faster followers always wait for the leader). In terms of performance, Varan allows the leader to run at almost native speed, as it does not require the leader to synchronize with the followers. While Varan provides flexibility in terms of matching the sequences of system calls issued by different versions, it does not provide an easy expressive way to encode the differences in system call sequences that should be tolerated across versions.

In this paper, we propose a simple, elegant, and expressive domain-specific language (DSL) specifically designed for writing system call matching rules that allows a follower to reconcile its sequence of system calls with that of the leader (§3). We show that this DSL allows the use of MVE in a wider range of scenarios with mini-

mal effort, requiring only a small number of rules in each case. In particular, we show the applicability of our approach with three different MVE scenarios: (1) running versions of the same application with different configurations (§2.1), where we needed only 7 rules to execute Redis under multiple configurations (§5.2), e.g., with and without a persistent store; (2) running different software revisions (§2.2), where we required only 7 rules to run versions of Redis which are up to 730 commits apart (§5.3); and (3) running native versions of an application in parallel with versions instrumented for dynamic analysis (§2.3), where we needed only 14 rules to support the Valgrind tool [24], 3 rules to support Asan [30], 1 rule to support Msan [33], and 4 rules to support Tsan [31] (§5.4).

We also provide an empirical evaluation that shows that simply comparing pairs of strace logs, which list the sequence of system calls that each version issues when run in isolation, is enough to write all the DSL rules (§5). No knowledge about the particular MVE system or the versions being used is needed to write the DSL rules. Inspired by how we manually found the rules, we provide an algorithm to synthesize some of the rules based on such pairs of strace logs (§4).

In summary, we make the following contributions:

1. The first paper to present a simple solution to the problem of handling divergent executions in MVE, which allows MVE to be easily applicable to many more scenarios, such as running an application concurrently under different configurations; running different releases of the same program; and running native versions in parallel with versions instrumented for dynamic analysis.
2. The design and implementation of a small and expressive DSL that encodes rules to handle divergences, and our experience using it in the three scenarios described above.
3. The design and implementation of an algorithm that synthesizes part of the DSL rules using pairs of strace logs, which can be obtained by running each version in isolation over the same inputs.
4. An empirical evaluation of our prototypes for the Varan MVE, that shows the applicability of each scenario and provides evidence about the little effort required to write the rules, and how much this task can be further automated by the DSL synthesis algorithm.

2 Applicable Scenarios

At a high-level, some program executions can be considered equivalent even if they do not execute the same code. As a trivial example, two executions of the same correct deterministic C program under different memory allocators can be considered equivalent because their *observable behavior*—the sequence of system calls they issue—is the same. However, there are scenarios in which it is beneficial to deploy programs with MVE that issue different sequences of system calls. For instance, one may increase reliability by deploying two releases of the same program [18] in which the order of some system calls are changed, but without affecting the overall behavior of the program—e.g., one release may simply change the order in which two files are opened.

In this paper, we describe a domain specific language (DSL) designed to easily encode and tolerate such divergences, and thus enable many useful MVE scenarios. In the rest of this section, we present and motivate three scenarios that can take full advantage of our DSL.

2.1 Different Configurations

Depending on its configuration parameters, software can behave differently by enabling or disabling features such as logging. For instance, Redis[1] is an in-memory key-value store that can optionally dump the store to persistent storage periodically or after every request.

There are three scenarios in which running different program configurations under MVE can be useful. First, for increased reliability: Different configurations may trigger different bugs so running several configurations in parallel increases the chance of at least one configuration staying alive and providing service. Second, for increased security: If security is critical, one may choose to stop as soon as any configuration diverges in its core execution from the others. The rationale here is that an attack may succeed in one configuration, but not all, as different configurations have slightly different memory layouts, issue different sequences of system calls, etc. Third, for inexpensive logging and error diagnosis: A fast configuration (no logging, no debugging info, full compiler optimizations, etc.) can be deployed at full speed, as the leader, while slower configurations (with logging, debugging, etc.) can be deployed in the background as followers.

Different configurations share the core functionality of the program, but each implements additional features such as logging and persistent storage. From the perspective of their external behavior, the sequence of system calls issued by an expensive configuration is typically a superset of the base configuration. For instance, the Redis configuration that adds persistence issues extra system calls to open the persistent file on disk and write data into it. In particular, one will see additional calls as below, interleaved with the core functionality of the program:

[1] `https://redis.io/`

```
1 ...
2 open("persistentStore", ...) = 7
3 ...
4 write(7, ...) = 10
5 ..
6 write(7, ...) = 45
7 ...
```

As we show in §3, our DSL makes it easy to encode such divergences, allowing MVE systems to run multiple configurations of the same program concurrently.

2.2 Different Software Releases

MVE is an effective technique to increase the reliability of software updates [11,18]. Instead of updating the software to a new version that becomes available, the idea is to run both the new and the old version in parallel. If one version fails, the system can revert to the other version. This technique mitigates the problem of unreliable software updates [14, 28, 36], as the old version is still available in the background in case the new version crashes.

Mx [18] applied this approach successfully, but it could deploy only versions that issue the same sequence of system calls. However, as we show in this paper, tolerating certain classes of system call divergences allows one to handle a much wider range of software updates.

In general, the external behavior of the software is stable, especially in mature applications. However, small changes in the sequence of system calls occur even for mature applications. Examples include: (1) slightly changing the API used, and (2) changing the order in which some system calls are performed. As an example in the first category, Lighttpd revision 2436 changes its sequence call sequence from `geteuid`, `geteguid` to `geteuid`, `getuid`, `getegid`, `getgid` [19]. As an example in the second category, Redis version 2.0.1 reorders the sequence `setsockopt`, `time`, `epoll_ctl` into `setsockopt`, `epoll_ctl`, `time`.

Our DSL makes it easy to express such differences. As we show in §5.3, we were able to run together Redis versions up to 730 commits apart while only using a small number of simple DSL rules.

2.3 Native and Sanitized Versions

Dynamic analysis techniques instrument or interpret the program under analysis to detect common programming errors. For instance, Asan [30], the address sanitizer that ships with modern C compilers, instruments memory buffers in the program with red zones to detect buffer overflow errors. Valgrind [25], a dynamic analysis tool that takes program binaries as input, interprets the program and shadows all the memory that the program uses to detect a large category of bugs, such as buffer overflows and invalid uses of uninitialized memory.

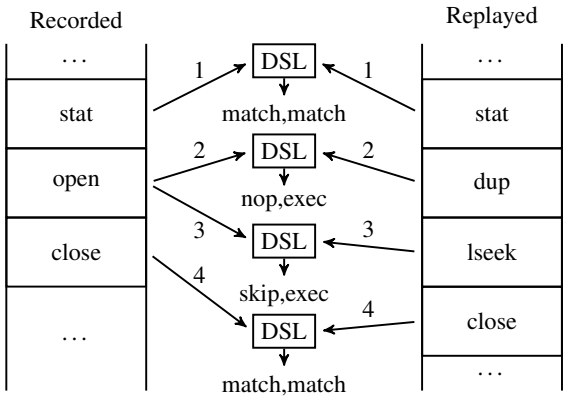


Figure 1: Example of reconciling two divergent sequences of system calls: `open` on the recorded side matches `dup` and `lseek` on the replayed side.

MVE can be used to deploy sanitized versions in the background [19]. The key idea is to run the native version of the program as a leader—this is the version that interacts with users and runs at full speed. Sanitized versions are then run as followers, checking the execution for errors in the background.

One of the main challenges involved is that the sanitized versions change the sequence of system calls that the program under analysis issues. For instance, analyses may use signal `SIGSEGV` internally (e.g., to allocate more shadow memory). This signal may also be used by the program under analysis. In this case, the analysis technique registers its own signal handler and intercepts attempts from the program under analysis to register another signal handler, through system call `rt_sigaction` in 64-bit GNU/Linux, by forwarding signals generated outside the analysis to the program's handler. We describe some of the changes in further detail in §3.1.

As we show in §5.4, our DSL is able to encode the divergences introduced by real dynamic analysis techniques, such as Asan [30], Msan [33], Tsan [31] and Valgrind [25], using only a small number of easy-to-write rules.

3 DSL for Reconciling MVE Divergences

We now propose a simple and expressive domain specific language (DSL) for describing system call divergences between two executions. Our design is driven by real-world examples illustrating the scenarios described in §2.

Figure 1 shows the high-level architecture of the DSL we propose. The DSL operates between two sequences of system calls: the *recorded* and the *replayed*. At each step, and for each sequence, the DSL takes as input the next system call and generates as output the ac-

```
dsl  ::= [#include "file"] [rule]
rule ::= [lhs-syscall] => [rhs-syscall]
       | begin => [rhs-syscall]
       | group g = { [ name ( [arg] ) ] }
lhs-syscall ::= scall pred lbl | nothing
rhs-syscall ::= scall pred callback
              | scall * | label | nothing

scall ::= name ( [arg] ) | g
pred  ::= { C-code } | ε
lbl   ::= as label | ε
callback ::= -> ( ret ) { C-code }
           | -> { C-code } | ε
name  ::= read | write | ...
arg   ::= _ | var
```

Figure 2: Syntax of the DSL. All words in bold and symbols besides `|`, `[`, and `]` are terminals. Square brackets denote possible empty comma-separated lists. *var*, *ret*, *label*, *file* and *g* are identifiers.

tion to take. For each matching system call between the recorded and replayed sequences (steps 1 and 4 in Figure 1), the DSL simply matches both sides through action MATCH, thus advancing both sequences by one position. In the example shown, taken from Valgrind, system call `open` is rewritten as a sequence of `dup` and `lseek`. The DSL reconciles the divergence in steps 2 and 3 through actions EXEC on the replayed sequence, to execute those system calls. In step 2, the recorded sequence is left unchanged through action NOP, while in step 3 it is advanced without matching anything on the replayed side through action SKIP.

3.1 Syntax

The syntax of the DSL is given by the grammar shown in Figure 2. A DSL input file is a collection of *rules*. Each rule defines how a sequence of recorded system calls, on the *left-hand side (LHS)* of the rule, matches a different sequence of replayed system calls, on the *right-hand side (RHS)* of that rule.

For instance, Figure 3a shows a rule that tolerates the divergence presented in Figure 1. The underscore characters allow any values for the respective arguments, so the rule matches a recorded `open` with a replayed sequence of `dup` and `lseek`, regardless of any arguments. For system calls where all arguments are unconstrained, we sometimes use a single underscore for brevity.

The RHS of each rule can refer to system calls on the LHS through *labels*. For instance, different releases of Redis register a different number of signal handlers in different order. The rule in Figure 3b shows how to use labels to reconcile such divergences.

Valgrind wraps 19 different system calls with the same three system calls before and three after. The user thus needs to repeat the same rule for each wrapped system

```
open(_,_,_) => dup(_), lseek(_,_,_)
```
(a)

```
rt_sigaction(_,_) as segv,
rt_sigaction(_,_) as ill,
rt_sigaction(_,_) as bus,
rt_sigaction(_,_) as fpe =>
 ill, rt_sigaction(_,_), bus, fpe, segv
```
(b)

```
group calls = { read(_,_,_), write(_,_,_) }

calls as self =>
 gettid(), write(_,_,_), rt_sigprocmask(_),
 self,
 rt_sigprocmask(_), gettid(), read(_,_,_)
```
(c)

```
open(p,_,_) {
  return !strcmp($(p), "overcommit");
} , read(_,_,_), close(_) => nothing
```
(d)

```
#include "globals.h" // declares ign

nothing => open("log.txt",_,_)
    -> (ret) { ign = $(ret); }

nothing => write(fd,_,_)
    { return $(fd) == ign; }
```
(e)

```
write(_,_,_) => write(_,_,count)
    -> (ret) {$(ret) = $(count)}
```
(f)

```
begin =>
 mprotect(_), mprotect(_) * , munmap(_)
```
(g)

Figure 3: Examples of DSL rules.

call. Instead, the DSL supports *groups*: syntactic sugar to repeat the same rule for different system calls. Figure 3c shows an example, adapted from Valgrind, that groups all the system calls that use the same rule.

Some rules apply only when particular values are passed to those system calls at runtime. For instance, when the memory allocator *malloc* reaches a certain percentage of the available memory, it tunes its behavior based on the kernel overcommit settings by reading file `/proc/sys/vm/overcommit_memory`. The analyses Asan and Tsan increase the virtual memory to a large percentage of the whole available addressing space, which prevents the allocator from ever tuning its behavior. The rule in Figure 3d reconciles such executions between native and sanitized versions, through a *predicate written as C code*.

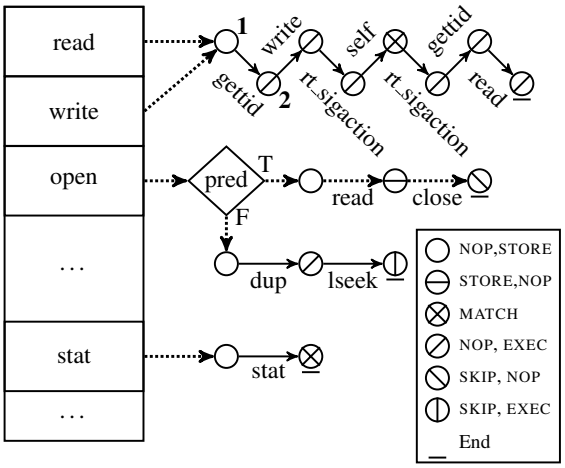


Figure 4: Visual representation of the DFAs generated for the rules in Figures 3c, 3d, and 3a, in that order. System call `stat` shows the default rule. Dotted and solid arrows transition on the LHS and RHS system calls, respectively. Node shapes denote actions performed on the recorded and replayed sequences on each transition.

The rule in Figure 3d also shows how to handle extra system calls that the replayed side issues without any correspondent on the recorded side through keyword `nothing` on the RHS. Of course, keyword `nothing` can also be used on the LHS. For instance, when deploying two versions of Redis in which the replayed side performs logging that the recorded side does not, we need a rule to ignore opening and writing to the log file. Figure 3e shows two such rules for `open` and `write`.

The rule in Figure 3e also shows an example of syntactic sugar for the common case of argument variables being literal strings or numbers. In this case, the DSL simply expands such a rule to an equivalent rule that uses predicates, similar to the rule shown in Figure 3d.

Predicates execute before a system call is matched, and therefore cannot access the results of that system call. Rules can also have a *callback* written in C, that executes after matching the system call and can access the (potentially modified) arguments and the return value. For instance, the rules shown in Figure 3e save the file descriptor of the matched `open` system call in global variable `ign`, which the rule for `write` then uses to discard the appropriate system calls. Note that the DSL allows the inclusion of C header files, which enables the rules to access libraries.

Predicates and callbacks can modify the arguments passed to the system call, and callbacks can also modify the return value. For instance, a version of Redis writes more bytes due to a protocol change. The rule shown in Figure 3f tolerates such a divergence by changing the return of the offending `write` on the replayed side.

The last part of the DSL, the `begin` rule, expresses a pattern that denotes the end of a large divergent prefix on the replayed side. System call matching using the other rules between both sides only starts after the pattern in the `begin` rule has been matched (which is empty by default). For instance, Valgrind sets up its internal state before starting to execute the program under analysis. This is when Valgrind sets handlers for interesting signals such as `SIGSEGV`, as discussed earlier. Valgrind finishes its set-up with one or more `mprotect` calls and a single `munmap` call. The rule in Figure 3g tolerates this large divergence in a compact way. This last example also shows the usage of the *star modifier (*)*, applied to the second `mprotect`, which matches the preceding system call zero or more times.

3.2 Semantics

The rules are implemented by a collection of *Deterministic Finite Automata (DFAs)*. Figure 4 shows the DFAs generated for some of the examples discussed in §3.1.

The algorithm starts by matching the current recorded system call with the first LHS on a rule to select a DFA. System calls that do not appear on the LHS of any rule have a default rule to themselves (e.g., rule `stat(_, _) as self => self` shown in Figure 4).

Rules are chosen in the same order in which they are defined. For instance, Figure 4 defines two rules that apply to system call `open`: the rules in Figures 3d and 3a, in that order. If the predicate is true, the algorithm chooses the first rule, otherwise it chooses the second one.

With a DFA selected, the algorithm uses it to match the rest of the sequence of system calls on the LHS of the rule, if any, with the sequence on the RHS. The DFA takes each system call to be reconciled, and either accepts it by moving to the next state, or rejects it. When the DFA rejects a system call, we say that an *irreconcilable divergence* has occurred because the two executions have diverged in a way that the DSL cannot reconcile. The DFA finishes once it accepts the final system call of a rule. The algorithm then discards the current DFA and uses the next recorded system call to select the next rule.

Rules that have `nothing` as the LHS are implemented as an exception to sequences rejected by the DFA. When the DFA rejects the first RHS system call of a rule with a single LHS, the algorithm then looks for a rule with `nothing` as the LHS that start with the offending RHS system call. If found, the algorithm follows that DFA instead of diverging. Otherwise, it diverges as described above. Rules with `nothing` on the LHS thus have lower precedence than all the other rules. Rules that have `nothing` as the RHS generate a DFA that only takes recorded system calls, as shown in Figure 4 for system call `open` when the predicate is true.

3.3 Interface with the MVE System

We now describe how an MVE system interacts with the DSL, using the example in Figure 1. Initially, the MVE system uses function `init` with the next recorded system call (e.g., `stat`) to choose a DFA. As suggested by Figure 4, a lookup table maps the first recorded system call to the DFA that implements the corresponding rule.

The MVE system then uses function `reconcile` to pass each recorded and replayed system call to the DSL. This function takes the current DFA, validates the next transition, and returns: the next DFA state, the actions to perform on the recorded and replayed sides, and some flags. On our running example, calling `reconcile(stat, stat)` yields actions MATCH on both sides. The MVE thus matches the two system calls and their arguments, copying the results from the recorded to the replayed side, and advances both sequences by one. Note that this behavior is what the MVE system does during regular operation without our DSL. This call also returns a flag that signals the end of this rule, so that the MVE uses function `teardown` to clean the resources of the finished DFA.

Following Figure 1, the next recorded call is `open`. Again, the MVE system uses function `init` to select the next rule. In this case, there is a choice between the rules in Figures 3d and 3a (defined in this order), depending on the truth value of the predicate in 3d. Let us assume that the predicate for 3d returns false, thus selecting the rule in 3a. At this point, calling `reconcile(open, dup)` yields actions NOP and EXEC on the recorded and replayed sides, respectively. Action NOP does nothing on the recorded side, while action EXEC executes the replayed system call without matching it with the recorded side. The MVE thus allows the replayed side to execute system call `dup` directly and calls `reconcile` with the same recorded call, `open`, and the next replayed call, `lseek`. Calling `reconcile(open, lseek)` returns actions SKIP and EXEC, for the recorded and replayed side, respectively. Action SKIP advances the recorded side one position, effectively ignoring the system call.

Let us now consider that the predicate for `open` returns true, selecting the rule in Figure 3d. In this case, calling `reconcile(open, read)` returns actions STORE and NOP for the recorded and replayed side, respectively. Action NOP on the replayed side means that the MVE system calls `reconcile` with the same replayed call, just as it does for NOP on the recorded side. Action STORE on the recorded side is useful for rules with multiple LHS calls, and prompts the MVE to advance the recorded side and call `reconcile` with the next recorded call. Later MATCH actions may refer to previous calls on the recorded side on which action STORE was taken, which means that the MVE needs to save all such recorded calls. For instance, the rule in Figure 3b returns the following sequence of actions for the recorded side: STORE, STORE, STORE, MATCH *2*, NOP, MATCH *3*, MATCH *4*, and MATCH *1*. Note that function `init` implicitly performs action STORE on the recorded side.

Function `reconcile` returns a special flag when an irreconcilable divergence occurs. The MVE must handle such a divergence, by reconciling it in some other way, stopping execution, or terminating that replayed version.

4 Automatic Synthesis of DSL Rules

When designing the DSL and writing rules for the different scenarios, we used *strace*,[2] an utility that logs all the system calls that a process issues, to generate system call traces for different program versions. For instance, we used this approach to compare the sequences of system calls issued by native and Valgrind versions of the same application when run on the same inputs.

We then noticed that a simple visual diff tool (*vimdiff*, part of VIM[3]) was able to display the two files side-by-side with most of the matching system calls aligned. Figures 5a and 5b show an example of such an aligned diff result. This provides empirical evidence that the rules are often easy to identify.

Based on our experience of manually writing the rules, we decided to create an automatic synthesis algorithm which targets the most common set of rules that we encountered, those of the form shown in Figure 3c, which wrap a system call with zero or more system calls before and zero or more system calls after. Ignoring grouping, 61% of the rules needed for Valgrind had this form.

Figure 5c shows the pseudo-code of the rule synthesis algorithm.[4] Function `synthesize` takes as input two system call traces, `lhs` and `rhs`, and returns a set of candidate rules. A candidate rule `Cand` is a triple `(before, s, after)` which defines the rule `s => before s after`, where `s` is a system call, and `before` and `after` are (possibly empty) sequences of system calls.

The algorithm iterates over each unique system call `s` in `lhs` (line 10). It aligns both sequences on the next instance of `s` on line 11, and creates an *initial candidate* `cand` by taking the `n` system calls around the aligned `s` on `rhs`. For instance, for `s=open` and `n=6`, the algorithm aligns both logs on position 56 and proposes rule: `open as s => access, getpid, getpid, gettid, write, rt_sigprocmask, s, rt_sigprocmask, gettid, read, fstat, mmap, fstat`.

[2] `https://strace.io/`

[3] `http://www.vim.org`

[4] For space reasons, the pseudo-code ignores error handling, particularly when function `align` fails.

```
50 access("ld.so.preload")
51
52
53
54
55
56 open("/etc/ld.so.cache")
57
58
59
60 fstat(3, ...)
61 mmap
62
63
64
65
66 close(3)
67
68
69
70
71
72 open("/lib/libm.so.6")
73
74
75
76
```

(a)

```
50 access("ld.so.preload")
51 getpid
52 getpid
53 gettid
54 write
55 rt_sigprocmask
56 open("/etc/ld.so.cache")
57 rt_sigprocmask
58 gettid
59 read
60 fstat(3, ...)
61 mmap
62 fstat
63 readlink
64 stat
65 mmap
66 close(3)
67 getpid
68 getpid
69 gettid
70 write
71 rt_sigprocmask
72 open("/lib/libm.so.6")
73 rt_sigprocmask
74 gettid()
75 read
76
```

(b)

```
1  // Helper functions:
2  // uniqueSCalls([[b,b,s,s,a,a,a] = [a,b,s]
3  // split(s, [b,b,s,s,a,a] = ([b,b], s, [s,a,a])
4  // takeHead([a,b,c,d],3) = [a,b,c]
5  // takeTail([a,b,c,d],3) = [b,c,d]
6  // intersectHead([a,b,c], [a,b,z]) = [a,b]
7  // intersectTail([b,c,d], [z,c,d]) = [c,d]
8
9  synthesize(Trace lhs, Trace rhs, int n) -> [Cand]
10  for s in uniqueSCalls(lhs) :
11   (lhs`, rhsBefore,  rhs`) = align(s, lhs, rhs)
12   before = takeTail(rhsBefore, n)
13   after = takeHead(rhs`, n)
14   cand = (before, s, after)
15   cands[s] = refineCand(cand, lhs`, rhs`)
16  return cands
17
18 align(SCall s, Trace lhs, Trace rhs)
19       -> (Trace, Trace, Trace)
20  (_, l, lhsAfter) = split(s, lhs)
21  while (rhs != []):
22   (rhsBefore, r, rhsAfter) = split(s, rhs)
23   if (l.args == r.args) :
24    return (lhsAfter, rhsBefore, rhsAfter)
25
26 refineCand(Cand c, Trace lhs, Trace rhs) -> Cand
27  while (lhs != [] and rhs != []):
28   (lhs, rhsBefore, rhs) = align(c.s, lhs, rhs)
29   before = intersectTail(c.before, rhsBefore)
30   after = intesectHead(c.after, rhs)
31  return c
```

(c)

Figure 5: Example sequences of system calls issued by a native execution (5a) and the same execution under Valgrind (5b), and pseudo-code for the synthesis algorithm (5c). Matching system calls in 5a and 5b are aligned and highlighted.

Of course, the initial candidate rule is unlikely to be correct. The algorithm then refines that candidate using the rest of the logs on line 15 as follows. First, it finds the next aligned pair of the same system call on line 28. In our example, this yields position 72 in the traces. The algorithm then computes the intersection of the current rule with the system calls that surround the new matching on lines 29–30. The algorithm repeats this refinement step for each aligned pair of the same system call, iteratively discarding system calls that were captured by accident by the initial candidate. Back to our example, the algorithm discards positions 50 and 60–62, thus finding the correct rule: `open as s => getpid, getpid, gettid, write, rt_sigprocmask, s, rt_sigprocmask, gettid, read`.

In this case, the algorithm found the correct rule in a single refinement step, but this may not be always the case. For instance, if position 76 contained system call `fstat`, as position 60, then the algorithm would keep system call `fstat` as the end of the refined rule, resulting in an incorrect rule due to *over-capture*. The algorithm is prone to over-capture for under-represented system calls (e.g., those that only appear once or twice in the whole log) because the algorithm cannot refine them past the initial candidate(s). Sorting system calls by their frequency in function `uniqueSCalls` improves the quality of the results by leaving under-represented system calls to the end.

The algorithm is also prone to *misalignment*, when it aligns two system calls incorrectly and then generates the trivial rule `syscall as s => s`. In our experience, misalignment happens only due to *non-determinism* (e.g., user input timing). Note that the system call comparison, in line 23, already handles some non-determinism. For instance, two `write` system calls on the same file descriptor and with the same size are considered equal, regardless of the contents. Similarly, two `open` system calls in directory `/tmp` are considered equal, even if the files have different names. This allows to align executions that print the current time or the process ID, and executions that create temporary files with different names. Note also that a correct MVE system handles these and other sources of non-determinism that happen during runtime.

The algorithm also fails when the pattern for reconciling a given system call changes. This may happen based on the arguments passed to the system call (e.g., opening a special file uses a different rule). The DSL provides C predicates to handle such cases, but the synthesis algorithm cannot generate them.

5 Evaluation

This section describes the empirical evaluation of the DSL we propose for reconciling system call divergences across program executions. In particular, we evaluate the DSL for each of the application scenarios that we describe in §2, using Varan as the underlying MVE system [19]. We also describe the empirical evaluation of the DSL synthesis algorithm in generating rules for the sanitized versions scenario.

5.1 Implementation

We implemented the DSL parser and the synthesis algorithm in Haskell, with 1388 and 422 LOC, respectively.

We evaluated the DSL with Varan as the MVE system, whose architecture we presented in the introduction. We modified Varan to work with the DSL in several ways. First, Varan builds C files from DSL input files and includes the generated files during compilation. At runtime, we added a flag for Varan to load a particular DSL file for the execution. The DSL-based matching runs on the followers. In our experiments, we use a single follower, but in principle we could run multiple followers, each with its own DSL rules.

We include the generated C files with Varan at build time for ease of implementation. There is no fundamental reason to prevent each DSL file to be compiled separately (e.g., as a shared object) and loaded dynamically. This would make Varan easier to extend to other scenarios and we plan to implement this feature in the future.

Currently, we implemented rules with multiple LHS system calls through several rules with a single LHS, bound together through predicates that keep track of the matching sequence. In the future, we plan to implement this directly in the DSL.

5.2 Different Configurations

We explored the scenario in §2.1 by deploying Redis 3.2.6 with Varan using different configurations. The leader was configured to keep an in-memory store and write minimal logs. The follower used one of the following configurations: (1) persistent store,[5] (2) verbose (debug) logs, and (3) both 1 and 2. We required only 7 rules to handle all divergences.

As expected, the sequences of system calls issued in these three configurations were a superset of those issued by the leader. Most of the DSL rules simply ignored extra operations performed over file paths. For instance, Configuration 1 required a variation of the rule shown in Figure 3e to ignore manipulating the persistence file.

[5] `appendonly yes` and `appendfsync always`.

Table 1: Redis versions tested, number of commits between versions, and number of rules needed.

ID	Versions	Commits	Rules
1	1.3.8* – 1.3.10	40	0
2	1.3.10 – 1.3.12	105	0
3	1.3.12 – 2.0.0	92	1
4	2.0.0 – 2.0.5	34	1
5	2.0.5 – 2.2.0	730	3
6	2.2.0 – 2.2.15	110	2

* Revision `a71f072`

We wrote a C library to simplify managing ignored file paths and associated descriptors to minimize the C code needed for each rule.

Configuration 1 issues one less `gettimeofday` call early in the execution. To reconcile this divergence, we had to use a long rule that captures context from the previous 8 system calls. Configuration 2 issues strictly more `gettimeofday` to write timestamps on log entries. A simple rule `nothing => gettimeofday(_,_)` sufficed to tolerate such divergences. Configuration 3 simply required a trivial merge of the DSL files for Configurations 1 and 2. However, we note that in general, merging DSL files is not guaranteed to tolerate the combination of behaviors that each file tolerates in isolation.

5.3 Software Releases

As discussed in §2.2, the DSL can be used to deploy different program releases. We deployed the pairs of Redis versions listed in Table 1, by running the old version as the leader and the new version as the follower. We configured leader and follower to use separate log files, with a verbose logging level. We then added rules for ignoring log files, with 6 rules totaling 15 lines. These rules are common to all experiments and are not included in Table 1. We used the *redis-benchmark* included in Redis 1.3.8 as our workload, configured with a single client and performing one request for each operation (we are interested in functionality rather than performance).

We start with Redis 1.3.8 revision `a71f072` so that our results can be compared with Mx, which could not deploy different versions that change the sequence of system calls [18]. Pairs 1 and 2 required no additional rules.

In Pair 3, version `2.0.0` registers one more signal handler than previous versions (for `SIGTERM`), which can be expressed with the simple rule `nothing => rt_sigaction(15,_,_)`. In Pair 4, version `2.0.5` changes the order of a `time` system call, from before to after an `epoll_ctl`. We used a rule similar to the rule shown in Figure 3b to tolerate this divergence.

Pair 5 had most changes, with 730 commits. However, all divergences introduced in these commits required only three rules, similar to those presented before: ignoring file paths and `stat` calls on those paths (one rule); issuing extra `time` system calls on one side (one rule); and a `write` system call that writes more bytes than previous versions, due to protocol differences, as we described in detail in §3.1 and Figure 3f (one rule).

Finally, pair 6 required the rule `nothing => gettimeofday(_,_)` to handle extra system calls, and a rule to tolerate a change in the order of multiple `rt_sigaction` system calls, as shown in Figure 3b.

We were able to deploy six pairs with releases up to 730 commits apart with minimal effort (7 rules in total). Our approach works especially well for applications that keep backwards-compatibility, such as Redis, which tend to retain external behavior between releases and newer versions still support older data formats and protocols.

5.4 Dynamic Analyses

We used Varan to deploy the following existing dynamic analyses as followers of a native leader: Asan, Msan, Tsan, and Valgrind (with the memcheck tool). Asan [30], Msan [33], and Tsan [31] are the address, memory, and thread sanitizer, respectively, which ship with modern releases of popular C/C++ compilers Clang and GCC. We used the ones that ship with Clang version 3.8. Valgrind [32] checks for uses of invalid memory (i.e. uninitialized, unallocated, or freed memory) in C/C++ programs through heavyweight dynamic instrumentation. We used Valgrind version 3.11 built from revision 15920 (VEX revision 3233).

We executed Git[6] version 2.9.2, a widely-used version control system, with all the analyses described above (commands `log`, `blame`, `diff`, and `tag`). We also executed the following applications with Asan and Valgrind: `ssh` and `ssh-keygen` from OpenSSH[7] version 7.1, a suite of utilities used to secure communications by encrypting network traffic; HTop[8] version 2.0.1, an interactive system monitor and process viewer, and VIM version 7.4, a screen-oriented text editor.

We manually wrote a DSL file for each analysis, making it possible to run all the programs we mentioned with all the configurations we listed. Msan required the smallest DSL file, with 1 rule totaling 7 lines; Asan required 3 rules totaling 10 lines, and Tsan required 4 rules totaling 13 lines. The most interesting rule, shared by all these three analyses, is shown in Figure 3d and described in §3.1. Other rules ignore the system calls in which the leader sets up signal handlers for signals that the analyses already handle, as described in §2.3; and ignore extra system calls that the analyses issue (`nanosleep` and `sched_getaffinity`).

As expected, Valgrind required more effort with 14 rules totaling 104 lines. Valgrind required a `begin` rule with 3 lines to ignore its initialization (rule in Figure 3g). We also grouped 19 system calls that use one of two rules (one shown in Figure 3c), thus saving implementation effort and improving the readability of the DSL file.

Handling system call `open` under Valgrind required two rules: a general rule, illustrated by positions 51–59 in Figure 5; and a specialized rule for file `/proc/self/cmdline`, the one in Figure 3a but with an appropriate predicate. When a process attempts to read the command line that launched it, Valgrind hides the fact that the process is being run under analysis by treating that `open` system call in a particular way instead of interpreting it directly. The specialized rule appears before the general rule in the file and, as explained in §3.3, has higher precedence.

We also needed some rules with `nothing` as the LHS. For instance, when the application under analysis attempts to `mmap` a file into memory, Valgrind issues more system calls to allocate adequate shadow memory for that file. Given that Varan does not copy the `mmap` system call to the ring buffer, we tolerated the divergence between positions 61–65 in Figures 5a and 5b with the following rule:

```
1 nothing => mmap(_), fstat(_,_,_),
2   readlink(_,_), stat(_,_), mmap(_)
```

Other rules with `nothing` as the LHS skip extra work that Valgrind performs to schedule threads, and when the program under analysis loads a dynamic library.

5.5 Synthesis Algorithm

All the rules described so far in the experimental evaluation were manually written by comparing sequences of strace logs side-by-side, as shown in Figure 5. We evaluated the DSL synthesis algorithm by using similar strace logs as input, to infer the rules needed to tolerate divergences between native and Valgrind executions, and comparing them with the ones that we wrote manually. We used the workloads for Git, OpenSSH, and VIM described in §5.4. We also used four GNU/Linux command-line utilities: `ls` and `du` from `CoreUtils`[9] version 8.25, `grep`[10] version 3.0, `cal` from `util-linux`[11] version 2.29.2, and the DSL synthesis algorithm itself. The algorithm took under 22 seconds on a modern laptop to run on each pair of traces.

[6] https://git-scm.com/
[7] http://www.openssh.com/
[8] http://hisham.hm/htop/
[9] https://www.gnu.org/software/coreutils
[10] https://www.gnu.org/software/grep/
[11] https://www.kernel.org/pub/linux/utils/util-linux/

Table 2: Rules synthesized from pairs of native and Valgrind strace logs, including partial and incorrect rules due to under-represented system calls ([a]), over-capture ([b]), or misalignment ([c]).

Program	Rules	Correct	Partial	Incorrect
git tag	4	4	0	0
git diff	5	4	1[a]	0
git log	5	5	0	0
ls	6	5	1[b]	0
grep	5	5	0	0
cal	4	4	0	0
du	5	4	1[a]	0
keygen	5	4	1[b]	0
ssh	9	6	1[a]	2[c]
synth	6	6	0	0
vim	11	6	2[b]	3[c]

Table 2 shows the results for the 19 different rules that can be synthesized from these traces (this is the number of rules before any grouping is applied). The total of column *Rules* is not 19 because of duplicate rules. For instance, let us consider that a row lists 2 rules with system calls *A* and *B* on the LHS; and another row lists 3 rules, for system calls *A*, *C*, and *D* on the LHS. The total is thus 4 rules, one for each of *A*, *B*, *C*, and *D* on the LHS.

There are 16 rules generated correctly from at least one pair of strace logs (column *Correct*). In two cases, the algorithm generates rules that have too many system calls, as described in §4 (column *Partial*). For instance, when saving a file, VIM always issues system call `utime` (to get the file modification times) before `setxattrs` (to set the file attributes). As a result, the synthesized rule for `setxattrs` is always (incorrectly) prefixed with system call `utime`.

The incorrect rules were all due to misalignment, as explained in §4. The algorithm generated incorrect rules only for interactive programs due to their inherent non-determinism which affects our collection of system call traces: OpenSSH relies on random data, from both client and server; and VIM blinks the cursor a different number of times between executions.

Overall, these results are encouraging: The algorithm was able to generate most of the rules that it is designed to synthesize, simplifying the manual effort required.

6 Limitations and Future Extensions

The DSL is already expressive enough to support all the different scenarios that we present in this document. In this section, we identify the main limitations that we believe will need to be addressed to apply the DSL to additional scenarios.

Greedy rule matching. Currently, rules are matched greedily. As a result, rules cannot share a prefix of system calls on the LHS. We plan to explore alternative semantics to support this case.

Distant system call matching. Rules that match system calls separated by a large number of uninteresting system calls are long and require the DSL to keep all these system calls in memory. We plan to extend the DSL to support this scenario better.

Multithreading. In multithreaded programs, our implementation for Varan uses a separate DFA per thread. However, all DFAs use the same set of rules. Future versions of the DSL could include ways to map rules to specific threads.

Composing rules. Code blocks in the DSL cannot be combined. We plan to explore combining DSL rules and blocks of code, as the following example shows:

```
1 R1::{ return fd == 1; } R2::{ return fd == 2; }
2 nothing => write(fd,_,_) R1 || R2
```

Synthesizing from multiple pairs of traces. The synthesis algorithm struggles with under-represented system calls. Applying the algorithm to several pairs of traces would result in a higher count of the rare system calls.

Synthesizing predicated rules. When the pattern for a system call changes, the synthesis algorithm generates wrong rules due to misalignment. The algorithm can be extended by assigning an integer measure of confidence to each rule, which increases as the rule matches more system calls. After a threshold, refining yields a new rule instead of updating the current one.

Synthesizing more rules. The DSL synthesis algorithm only generates rules involving the original system call on the RHS. This algorithm can be extended to consider unmatched portions of the logs surrounded by matched sequences. For instance, for the divergence between lines 61–65 in Figures 5a and 5b, the rule presented in §5.4 can be extracted by considering the surrounding matched `fstat` and `close` in the strace logs.

7 Related Work

In this paper, we propose a DSL approach to reconcile system call divergences in the context of the multi-version execution [8,12,13,18,19,22,29,34,35]. Mx [18] performs dynamic software updates by running different program revisions in different versions. However, Mx can only deploy revisions that issue the same exact sequence of system calls; it does not tolerate any divergences. Varan [19] provides limited support for tolerating system call divergences through BPF filters [23] that rewrite the sequence of system calls. Varan was thus able

to deploy program revisions that Mx could not. However, the BPF filters support only a single system call on the left-hand side, and the filters are very difficult to write by someone not familiar with BPF. Tachyon [22] also supports rewriting the sequence of system calls between two processes. However, it does not keep state between invocations by design; and it is also limited to a single system call on the left-hand side. Both Varan and Tachyon do not support the reordering of several system calls on the left to several others on the right, as we do in Figure 3b. On the other hand, the DSL we propose vastly improves the support for reconciling divergences, allowing for sophisticated rules that trigger only under certain conditions, which keep state about the current divergence being handled, which provide better support for complex reordering of system calls, and which require less effort to write given the expressiveness of the DSL.

Each DSL rule is compiled to a DFA, which resembles how regular expressions are efficiently implemented [1]. In fact, a RHS rule without predicates or callbacks is a regular expression for an alphabet in which each symbol is a system call. However, adding predicates to accept each symbol conditionally and callbacks after each symbol is accepted is an important difference.

Andersen and Lawall [2] propose a DSL and an algorithm for specifying and inferring generic patches to C programs which capture the collateral evolution of library call sites when the API changes. Instead, our DSL and synthesis algorithm operates on system call traces, which present specific challenges and opportunities.

Bakken et al. [4] propose a DSL for describing how to combine votes of multiple versions into a single result. Our DSL focuses on matching, instead of combining, the two "votes" on the recorded and replayed sides.

Techniques for synthesizing regular expressions from examples are related to the technique we propose for generating DSL rules from strace logs. Approaches based on genetic algorithms designed in the context of text extraction [6,7,10] are useful for generalizing the observed behavior to unseen examples. However, they do not match our goal of generating rules that exactly match all observed divergences. Regular expressions and DFAs can be synthesized through techniques that require a set of positive and negative examples [3, 9, 21, 26]. These approaches are not directly applicable because part of the challenge of synthesizing DSL rules is to identify the positive examples, and the concept of negative examples does not apply directly to strace logs.

Program synthesis techniques [5,15,16,17,27] are also related to the technique we propose for generating DSL rules. The most relevant technique is λ^2 [15], which performs an enumerative search in a loop that generates candidates and refines them iteratively. Our technique generates a single candidate per rule using the first match and then refines it iteratively using the rest of the strace log. λ^2 can generate more than one candidate, and it uses a cost model to guide the program generation to generate a minimal cost solution. The extension we propose of adding a metric for the confidence of a candidate and keeping candidates above a threshold is similar.

8 Conclusion

In this paper, we have presented a simple and expressive domain-specific language (DSL) to write rules to tolerate expected divergences in the sequences of system calls issued by different program executions. The DSL we propose enables the deployment of multi-version execution (MVE) systems in a wider range of scenarios. In particular, we showed its applicability to three scenarios: (1) running versions of the same program under different configurations; (2) running different software releases, and (3) running native programs together with versions instrumented for dynamic analysis.

We report the results of an experimental evaluation for all the scenarios by manually writing the required rules starting from pairs of system call trace logs, obtained for each version in isolation. In particular, we show that the user needs no knowledge about the internals of the programs and analyses being deployed through MVE. We provide empirical evidence of the low effort required to identify and write such DSL rules, and we present the design and evaluation of an algorithm to automatically extract some of the DSL rules from such pairs of logs.

We believe that the DSL we propose, with its ability to easily encode divergences in the sequences of system calls issued by two executions, is an important contribution that will enable exciting new research on multi-version execution.

9 Acknowledgments

We thank our shepherd Gilles Muller, the anonymous reviewers, and Andrea Mattavelli and for their useful feedback on the paper. We also thank the EPSRC for sponsoring this research through an Early-Career Fellowship and a CDT studentship.

References

[1] AHO, A. V., LAM, M. S., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison Wesley, 2006.

[2] ANDERSEN, J., AND LAWALL, J. L. Generic patch inference. In *Proc. of the 23rd IEEE International Conference on Automated Software Engineering (ASE'08)* (Sept. 2008).

[3] ANGLUIN, D. Learning regular sets from queries and counterexamples. *Information and Computation 75*, 2 (Nov. 1987), 87–106.

[4] BAKKEN, D. E., ZHAN, Z., JONES, C. C., AND KARR, D. A. Middleware support for voting and data fusion. In *Proc. of the 2001 International Conference on Dependable Systems and Networks (DSN'01)* (July 2001).

[5] BAROWY, D. W., GULWANI, S., HART, T., AND ZORN, B. Flashrelate: Extracting relational data from semi-structured spreadsheets using examples. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'15)* (June 2015).

[6] BARTOLI, A., DAVANZO, G., DE LORENZO, A., MEDVET, E., AND SORIO, E. Automatic synthesis of regular expressions from examples. *Computer 47*, 12 (Dec. 2014), 72–80.

[7] BARTOLI, A., DE LORENZO, A., MEDVET, E., AND TARLAO, F. Inference of regular expressions for text extraction from examples. *IEEE Transactions on Knowledge and Data Engineering 28*, 5 (May 2016), 1217–1230.

[8] BERGER, E. D., AND ZORN, B. G. Diehard: probabilistic memory safety for unsafe languages. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'06)* (June 2006).

[9] BONGARD, J., AND LIPSON, H. Active coevolutionary learning of deterministic finite automata. *Journal of Machine Learning Research 6* (Dec. 2005), 1651–1678.

[10] BRAUER, F., RIEGER, R., MOCAN, A., AND BARCZYNSKI, W. M. Enabling information extraction by inference of regular expressions from sample entities. In *Proc. of the 22th ACM International Conference on Information and Knowledge Management (CIKM'11)* (Oct. 2011).

[11] CADAR, C., AND HOSEK, P. Multi-version software updates. In *Proc. of the 4th Workshop on Hot Topics in Software Upgrades (HotSWUp'12)* (June 2012).

[12] CHEN, L., AND AVIZIENIS, A. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. of the 8th IEEE International Symposium on Fault Tolerant Computing (FTCS'78)* (June 1978).

[13] COX, B., EVANS, D., FILIPI, A., ROWANHILL, J., HU, W., DAVIDSON, J., KNIGHT, J., NGUYEN-TUONG, A., AND HISER, J. N-variant systems: A secretless framework for security through diversity. In *Proc. of the 15th USENIX Security Symposium (USENIX Security'06)* (July-Aug. 2006).

[14] CRAMERI, O., KNEZEVIC, N., KOSTIC, D., BIANCHINI, R., AND ZWAENEPOEL, W. Staged deployment in Mirage, an integrated software upgrade testing and distribution system. In *Proc. of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)* (Oct. 2007).

[15] FESER, J. K., CHAUDHURI, S., AND DILLIG, I. Synthesizing data structure transformations from input-output examples. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'15)* (June 2015).

[16] GULWANI, S. Dimensions in program synthesis. In *Proc. of the 12th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'10)* (July 2010).

[17] GULWANI, S. Automating string processing in spreadsheets using input-output examples. In *Proc. of the 38th ACM Symposium on the Principles of Programming Languages (POPL'11)* (Jan. 2011).

[18] HOSEK, P., AND CADAR, C. Safe software updates via multi-version execution. In *Proc. of the 35th International Conference on Software Engineering (ICSE'13)* (May 2013).

[19] HOSEK, P., AND CADAR, C. Varan the Unbelievable: An efficient N-version execution framework. In *Proc. of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)* (Mar. 2015).

[20] KONING, K., BOS, H., AND GIUFFRIDA, C. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *Proc. of the 2016 46th International Conference on Dependable Systems and Networks (DSN'16)* (June 2016).

[21] LEE, M., SO, S., AND OH, H. Synthesizing regular expressions from examples for introductory automata assignments. In *Proc. of the 2016 International Conference on Generative Programming and Component Engineering (GPCE'16)* (Oct. 2016).

[22] MAURER, M., AND BRUMLEY, D. TACHYON: Tandem execution for efficient live patch testing.

In *Proc. of the 21st USENIX Security Symposium (USENIX Security'12)* (Aug. 2012).

[23] MCCANNE, S., AND JACOBSON, V. The BSD packet filter: A new architecture for user-level packet capture. In *Proc. of the 1993 Winter USENIX Conference (USENIX Winter'93)* (Jan. 1993).

[24] NETHERCOTE, N., AND SEWARD, J. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science 89*, 2 (2003).

[25] NETHERCOTE, N., AND SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'07)* (June 2007).

[26] PAREKH, R., AND HONAVAR, V. G. Learning dfa from simple examples. *Machine Learning 44*, 1-2 (July 2001), 9–35.

[27] PERELMAN, D., GULWANI, S., GROSSMAN, D., AND PROVOST, P. Test-driven synthesis. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'14)* (June 2014).

[28] PINA, L., AND HICKS, M. Tedsuto: a general framework for testing dynamic software updates. In *Proc. of the IEEE International Conference on Software Testing, Verification, and Validation (ICST'16)* (Apr. 2016).

[29] SALAMAT, B., JACKSON, T., GAL, A., AND FRANZ, M. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proc. of the 4th European Conference on Computer Systems (EuroSys'09)* (Mar.-Apr. 2009).

[30] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., AND VYUKOV, D. AddressSanitizer: A fast address sanity checker. In *Proc. of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)* (June 2012).

[31] SEREBRYANY, K., AND ISKHODZHANOV, T. ThreadSanitizer—data race detection in practice. In *Workshop on Binary Instrumentation and Applications* (12 2009).

[32] SEWARD, J., AND NETHERCOTE, N. Using Valgrind to detect undefined value errors with bit-precision. In *Proc. of the 2005 USENIX Annual Technical Conference (USENIX ATC'05)* (Apr. 2005).

[33] STEPANOV, E., AND SEREBRYANY, K. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proc. of the International Symposium on Code Generation and Optimization (CGO'15)* (Feb. 2015).

[34] VOLCKAERT, S., COPPENS, B., VOULIMENEAS, A., HOMESCU, A., LARSEN, P., SUTTER, B. D., AND FRANZ, M. Secure and efficient application monitoring and replication. In *Proc. of the 2016 USENIX Annual Technical Conference (USENIX ATC'16)* (June 2016).

[35] XUE, H., DAUTENHAHN, N., AND KING, S. T. Using replicated execution for a more secure and reliable web browser. In *Proc. of the 19th Network and Distributed System Security Symposium (NDSS'12)* (Feb. 2012).

[36] YIN, Z., YUAN, D., ZHOU, Y., PASUPATHY, S., AND BAIRAVASUNDARAM, L. How do fixes become bugs? In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'11)* (Sept. 2011).

Titan: Fair Packet Scheduling for Commodity Multiqueue NICs

Brent Stephens, Arjun Singhvi, Aditya Akella, Michael Swift
UW-Madison

Abstract

The performance of an OS's networking stack can be measured by its achieved throughput, CPU utilization, latency, and per-flow fairness. To be able to drive increasing line-rates at 10Gbps and beyond, modern OS networking stacks rely on a number of important hardware and software optimizations, including but not limited to using multiple transmit and receive queues and segmentation offloading. Unfortunately, we have observed that these optimizations lead to substantial flow-level unfairness.

We describe Titan, an extension to the Linux networking stack that systematically addresses unfairness arising in different operating conditions. Across both fine and coarse timescales and when NIC queues are undersubscribed and oversubscribed, we find that the Titan can reduce unfairness by 58% or more when compared with the best performing Linux configuration. We also find that improving fairness can lead to a reduction in tail flow completion times for flows in an all-to-all shuffle in a cluster of servers.

1 Introduction

Many large organizations today operate data centers (DCs) with tens to hundreds of thousands of multi-core servers [37, 35, 20]. These servers run a variety of applications with different performance needs, ranging from latency-sensitive applications such as web services, search, and key-value stores, to throughput-sensitive applications such as Web indexing and batch analytics. With the scale and diversity of applications growing, and with applications becoming more performance hungry, data center operators are upgrading server network interfaces (NICs) from 1Gbps to 10Gbps and beyond. At the same time, operators continue to aim for multiplexed use of their servers across multiple applications to ensure optimal utilization of their infrastructure.

The main goal of our work is to understand how we can enable DC applications to drive high-speed server NICs while ensuring key application performance goals are met—i.e., throughput is high and latency is low—and key infrastructure performance objectives are satisfied—i.e., CPU utilization is low and applications share resources fairly.

Modern end-host network stacks offer a variety of optimizations and features to help meet these goals. Foremost, many 10Gbps and faster NICs provide multiple hardware queues to support multi-core systems. Recent advances in the network stack (RPS [7]/RFS [6]/XPS [11]) allow systematic assignment of these queues and the flows using them to CPU cores to reduce cross-core synchronization and improve cache locality. In addition, provisions exist both in hardware and in the operating system for offloading the packetization of TCP segments, which vastly reduces CPU utilization [22]. Likewise, modern OSes and NIC hardware provide a choice of software queuing logics and configurable queue size limits that improve fairness and lower latencies by avoiding bufferbloat [19].

The first contribution of this paper is a systematic exploration of the performance trade-offs imposed by different combinations of optimizations and features for four key metrics, namely, throughput, latency, CPU utilization, and fairness. We study performance under extensive controlled experiments between a pair of multi-core servers with 10G NICs where we vary the level of oversubscription of queues.

We find that existing configuration options can optimize throughput and CPU utilization. But, we found that across almost every configuration there is substantial *unfairness* in the throughput achieved by different flows using the same NIC: some flows may transmit at twice the throughput or higher than others, and this can happen at both fine and coarse time scales. Such unfairness increases tail flow completion times and makes data transfer times harder to predict. We find that this unfair-

ness between flows arises because of three key aspects of today's networking stacks:

Foremost, OSes today use a simple hash-based scheme to assign flows to queues, which can easily lead to hash collisions even when NIC queues are undersubscribed (fewer flows than queues). Even a more optimal flow-to-queue assignment can result in flow imbalance across queues especially under moderate oversubscription (when the number of flows is only slightly larger than the number of queues).

Second, NIC schedulers strive for equal throughput from each transmit queue and thus service packets from queues in a strict round-robin fashion. Flows that share a queue as a result receive only a fraction of the throughput of those that do not. Even over long periods, a flow may receive half its fair-share throughput or less.

Finally, segmentation offload, which is crucial for lowering CPU utilization, exacerbates head-of-line blocking because a large segment of a flow must be transmitted before a segment from a different flow can be transmitted out of the same queue. This becomes acute at high levels of oversubscription, when there may be multiple segments from different flows in each queue. In this case, head-of-line blocking is also exacerbated by the number of queues that are in use. The NIC performs round robin scheduling of packets from different queues, and the OS aims to keep the same number of bytes enqueued in each hardware queue. If a large segment of the same size is in every queue, a newly arrived packet will have to wait for every enqueued segment to be sent before it can be sent, regardless of which queue it uses.

The second contribution of this paper is an extension to the Linux networking stack called *Titan* that incorporates novel ideas to overcome the above fairness issues. First, Titan uses *dynamic queue assignment* (DQA) to evenly distribute flows to queues based on current queue occupancy. This avoids flows sharing queues in undersubscribed conditions. Second, Titan adds a new queue weight abstraction to the NIC driver interface and a *dynamic queue weight assignment* (DQWA) mechanism in the kernel, which assigns weights to NIC queues based on current occupancy. In Titan, NICs use deficit round-robin [36] to ensure queues are serviced according to computed weights. Third, Titan adds *dynamic segmentation offload sizing* (DSOS) to dynamically reduce the segment size and hence reduce head-of-line blocking under over-subscription, which balances improvements to fairness against increased CPU utilization.

We implement Titan in Linux, and, using experiments both without and with network congestion, we show that Titan greatly reduces unfairness in flow throughput across a range of under- and oversubscription conditions and both at short and long timescales. In many cases, there is near zero unfairness, and in the cases where it remains, Titan reduces unfairness by more than 58%. Our experiments on a cluster of servers show that Titan offers the most fair flow completion times and decreases flow completion times at the tail (90th percentile).

Titan can increase CPU utilization and latency. We have designed Titan so as to try to minimize its impact on CPU utilization. In our experiments, Titan with DQA and DQWA often increases CPU utilization by less than 10%, although in the worst case it increases CPU utilization by 17% and 27% with and without pinning queues to cores, respectively. Also, Titan often matches the RTT latency of unmodified Linux with average latencies ranging from 123–660μs. At most, Titan increases latency by 134μs, and DSOS often reduces latency by more than 200μs. Still, latency under load still remains higher than when there is no other traffic using the NIC (32μs).

Current best practices for preventing long-running bulk data transfers from impacting latency sensitive traffic is to isolate different traffic classes in different priorities [26, 20]. Titan is compatible with DCB, so DCB priorities can still be used to isolate latency-sensitive traffic from bulk traffic in Titan. At the NIC level, this is accomplished by allocating dedicated pools of NIC queues for each DCB priority.

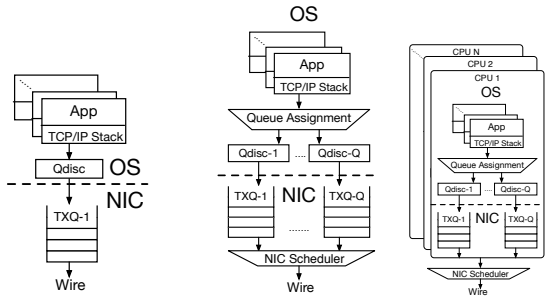
In the next section we provide background material on server networking stacks. Section 3 describes the design of Titan, and Section 4 has information on the implementation. Sections 5 and 6 describe our methodology and evaluation. We follow with related work and then we conclude.

2 Background

Networking in modern OSes is complex. There are multiple cooperating layers involved, and each layer has its own optimizations and configurations. Further, there are multiple different dimensions by which the performance of a server's network stack can be measured, and different configurations have subtle performance trade-offs. Figure 1 shows the different layers involved in a server's network stack (server-side networking), and Table 1 lists the most significant configuration options.

2.1 Server Networking Queue Configurations

We focus on the *transmit* (TX) side of networking because choices made when transmitting segments have a much larger potential to impact fairness: a server has no control over what packets it receives and complete control over what segments it transmits. Although the RX-side of networking is important, TX and RX are largely independent, so recent improvements to the RX



(a) **(b)** **(c)**

Figure 1: Different server-side TX networking designs: (a) Single queue (SQ) TX networking. (b) Multiqueue (MQ) TX networking. (c) Multicore-partitioned (XPS) multiqueue TX networking.

side [25, 18, 30] are complementary to improvements to the TX side.

In an OS, data from application buffers are passed as a segment (smaller than some maximum segment size) through many different layers of the network stack as it travels to the NIC, where it is turned into one or more packets on the wire. Both the design of each layer that touches a segment and the interfaces between them can impact performance.

There are many ways of connecting the layers of a networking stack that differ in the number of *NIC transmit queues* and *the assignment of queues to CPU cores*. Figure 1 illustrates three designs. Figure 1a shows how the OS interfaces with a single queue NIC (SQ). Figures 1b and 1c show two different ways for an OS to interface with a multiqueue NIC. The first (MQ) allows for flows on any core to use any NIC queue. The second partitions queues into pools that are dedicated to different cores, which we refer to by its name in Linux, XPS (transmit packet steering) [11].

Single Queue (SQ): In this design, segments from multiple competing applications (and containers/VMs) destined for the same output device are routed by the TCP/IP stack first to a per-device software queue and then to a per-device hardware queue (Figure 1a). The software queue (`Qdisc` in Linux) may implement any scheduling policy. The hardware transmit queues are simple FIFOs.

On a multicore system, SQ can lead to increased resource contention (locking, cache coherency, *etc.*). Thus, SQ has largely been replaced by designs that use multiple independent software and hardware transmit queues. Nevertheless, SQ offers the OS the most control over packet scheduling because the NIC will transmit packets in the exact order chosen by the OS.

Multiqueue (MQ): To avoid SQ's resource contention overheads, many 10 Gbps and faster NICs provide multiple hardware transmit and receive queues (MQ). Most OSes use multiple partitioned software queues, one for each hardware queue. Figure 1b illustrates MQ in Linux. Note that queues are not pinned to individual cores in this model, although flows may be assigned to queues. This allows computation to be migrated to idle or underutilized cores [32] at the expense of performance isolation provided by dedicating queues to cores. Given a multiqueue NIC, by default, Linux will use MQ.

The driver that we use (`ixgbe`) sets the number of queues to be equal to the number of cores by default. However, modern NICs typically can provide more hardware queues than cores, and using more queues than cores can be advantageous.

Moving to a multiqueue NIC requires that the OS implement some mechanism for assigning traffic to queues. In Linux, queue assignment is determined by RSS hashing for incoming flows and by a per-socket hash for outgoing flows. Because the number of open sockets may be much larger than both the number of NIC queues and the number of simultaneously active sockets, hash collisions would be expected given this approach regardless of the specific hash algorithm that is used.

In MQ, NICs must implement some algorithm for processing traffic from the different queues because they can only send a single packet at a time on the wire. Both the Intel 82599 and Mellanox ConnectX-3 NICs perform round-robin (RR) scheduling across competing queues of the same priority [2, 31]. Because of this, MQ can increase HOL blocking latency. If a multi-packet segment is enqueued in an empty queue, the time to send this entire segment in MQ will be the transfer time in SQ multiplied by the number of active queues. For example, sending a single 64KB segment at 10Gbps line-rate takes 52μs, while sending a 64KB segment from 8 different queues takes 419μs. Further, if all of the queues are full, the queueing latency of the NIC for any new segment is at least equal to the minimum number of bytes enqueued in a queue times the number of queues.

Multicore-Partitioned Multiqueue (XPS): The third networking design partitions NIC queues across the available CPUs, which can reduce or eliminate the inter-core communication performed for network I/O and improve cache locality. This configuration (transmit packet steering or XPS [11]) is particularly important for performance isolation because it ensures VMs/containers on one core do not consume CPU resources on another core to perform I/O. As in MQ, when a core can use multiple queues, hashing is used to pick which queue individual flows are assigned to in Linux.

In Linux, partitioning queues across cores involves significant configuration. XPS assigns NIC TX queues to a pool of CPUs. Because many TX queues can share an interrupt, interrupt affinity must also be configured correctly for XPS to be effective.

Config	Purpose	Expected Impact
Segmentation offloading (TSO/GSO)	Offload or delay segment packetization	Increases to segment sizes should reduce CPU utilization, increase latency, and hurt fairness
Choice of software queue (`Qdisc`)	Optimize for different performance goals	Varies
Assignment of queues to CPU cores (XPS, *etc.*)	Improve locality and performance isolation	Improved assignment should reduce CPU utilization
TCP queue occupancy limits (TCP Small Queues)	Avoid bufferbloat	Decreasing should reduce CPU utilization and latency up to a point of starvation.
Hardware queue occupancy limits (BQL)	Avoid head-of-line (HOL) blocking	Decreasing the byte limit should reduce latency up to a point of starvation. Further decreases should decrease throughput.

Table 1: A table that lists the different server-side network configurations investigated in this study, their purpose, and their expected performance impact.

2.2 Optimizations and Queue Configurations

There are many additional configurations and optimizations that impact network performance. Combined with the above queue configurations, these options induce key trade-offs in terms of latency, throughput, fairness and CPU utilization.

TSO/GSO: *Segmentation offloading* allows the OS to pass segments larger than the MTU through the network stack and down to the NIC. This reduces the number of times the network stack is traversed for a given bytestream. There are many per-segment operations in an OS networking stack, so increasing segment sizes reduces CPU utilization [28].

Many NICs are capable of packetizing a TCP segment without CPU involvement, called TCP Segmentation Offloading (TSO). For NICs that do not support TSO, Generic Segmentation Offloading (GSO) provides some of the benefit of TSO without hardware support by passing large segments through the stack and segmenting only just before passing them to the driver.

TSO/GSO hurts latency and fairness by causing HOL blocking. Competing traffic must now wait until an entire segment is transmitted. Further, sending large segments can cause bursts of congestion in the network [24]. To avoid the problems associated with TSO/GSO, Linux does not always send as large of segments as possible. Instead, Linux automatically reduces the size of TSO segments to try to ensure that at least one segment is sent each millisecond [9]. In effect, this causes Linux to use smaller segments on slow networks while still using as large of segments as possible on fast networks. (*e.g.* 10 Gbps and beyond).

Software Queue Discipline: Before segments are passed to a hardware queue, they are processed by a software queue (`Qdisc`). By default, the queuing discipline in Linux is FIFO (`pfifo_fast`), which is sub-optimal for latency and fairness. Linux implements at least two other superior policies: (1) The `prio` policy strictly prioritizes all traffic from a configurable class over all other traffic, improving latency. (2) The `sfq` policy implements Stochastic Fair Queueing (SFQ) using the deficient round robin (DRR) scheduling algorithm [36] to fairly schedule segments from competing flows regardless of differing segment sizes.

TSO Interleaving: Transmitting an entire TSO segment at once for a given queue can significantly increase latency and harm fairness, even if each queue is serviced equally. Some NICs address this with *TSO interleaving* [2, 31], which sends a single MTU sized packet from each queue in round-robin even if TSO segments are enqueued. This can lead to fairer packet scheduling as long as there is only one flow per-queue. HOL blocking can still occur if there are multiple flows in a queue.

TCP Queue Occupancy Limits: Enqueuing too many bytes for a flow into software queues causes bufferbloat [19], which can hurt latency and fairness. TCP Small Queues (TSQ) [10] limits the number of outstanding bytes that a flow may have enqueued in either hardware or software queues to address this problem. Once the limit is reached (256KB by default in Linux), the OS waits for the driver to acknowledges that some segments for that flow have been transmitted before enqueuing more data. As long as more bytes are enqueued per-flow than can be transmitted by the NIC before the next interrupt, TSQ can still drive line-rate while reducing bufferbloat.

In Linux, the enqueueing of additional data for flows sharing a queue in TSQ happens in batches. This is a side-effect of Linux using the freeing of an `skbuff` as a signal that it has been transmitted and `skbuffs` only being freed by the driver in batches in the TX interrupt handler.

Hardware queue occupancy limits: Hardware queues are simple FIFOs, so increasing the bytes enqueued per-hardware queue directly increases HOL blocking latency. Byte Queue Limits (BQL) [1] in Linux limits the total amount of data enqueued in a hardware queue. However, it is important to enqueue at least as many bytes as can be sent before the next TX interrupt, otherwise starvation may ensue. A recent advancement is Dynamic Queue Limits (DQL) [1], which dynamically adjusts each hardware queue's BQL independently so as to decrease HOL blocking while avoiding starvation.

2.3 Configuration Trade-off Study

We studied the impact of the aforementioned configurations on server-side performance (CPU utilization,

Cvanilla:	Default Linux networking stack incurs significant latency and unfairness, regardless of how many NIC queues are used, but has high throughput and low CPU.
C1: No TSQ:	TSQ is an important optimization. Disabling can cause significant latency and unfairness.
C2: Improved software scheduling:	Improving the software scheduler can significantly reduce latency and increase fairness, especially when only a single NIC queue is used. Comes at the cost of CPU utilization.
C3: No BQL:	BQL is an important optimization because disabling it can lead to increased latency and decreased fairness.
C4: 64KB BQL:	Setting BQL too small decreases latency but hurts fairness at long timescales with many flows.
C5: No TSO:	Disabling segmentation offloading hurts every performance metric because CPUs saturate.
C6: 16KB GSO:	Using a smaller GSO size than the default (64KB) improves fairness at short timescales (ms), increases CPU utilization.
Cmax: C2 + 256KB BQL:	Dynamic Queue Limits (DQL) leads to a higher queue limit than necessary to avoid starvation. If BQL is manually set smaller, it is possible to reduce latency and improve fairness.

Table 2: Summary of experimental results for different networking configurations.

throughput, latency, and fairness). Our high-level takeaways are listed in Table 2. These are synthesized from the raw results presented for each combination of workload, queue configuration, and optimization, which we detail in a technical report [40]. Table 3 in Section 6 shows the raw results for default Linux (*Cvanilla*) and the best performing configuration (*Cmax*). These results show that using SFQ for the queuing discipline with TCP small queues enabled and byte queue limits manually set to 256KB tend to out-perform all other combinations across different queue configurations. This is denoted by Cmax, which we henceforth focus on as the baseline best-performing MQ/XPS configuration today.

While we find that using multiqueue NICs can generally offer low CPU utilization and high throughput, we also find that the current Linux networking stack is unable to provide fairness at any time scale across flows at any subscription level. In the undersubscribed case, the central problem with MQ in Linux is the assignment of flows to queues. At low oversubscription, unfairness is uniformly high at short (1ms) and long (1 sec) timescales. We find that this largely occurs because some queues have more flows than others, and flows that share a queue send half as much data as those that do not. At high oversubscription, fairness is uniformly worse, as hashing is not perfect and leads to variable number of flows per queue, and a flow sharing a queue with 9 other flows will send much more slowly than one sharing with 5. However, using the best practices, exemplified particularly by configuration Cmax, can have substantial benefits over vanilla Linux without optimizations (Cvanilla).

2.4 Summary

Multiqueue NICs allow different CPU cores to perform network I/O independently, which is important for reducing the CPU load of network I/O caused by locking and cross-core memory contention. Each core can use independent software queueing disciplines feeding independent hardware queues. Further, TSO reduces CPU utilization by allowing the OS to treat multiple sequential packets as a single large segment. However, as a consequence, a packet scheduler in the NIC is now responsible for deciding which queue is allowed to send packets out on the wire. Because the NIC performs round-robin scheduling across competing hardware queues and TSO segments cause HOL blocking, the NIC will emit an unfair packet schedule when the network load is asymmetrically partitioned across the NIC's hardware queues and when multiple flows share a queue.

3 Titan

This section presents the design of Titan, an OS networking stack that that introduces new mechanisms for improving network fairness with multiqueue NICs. To improve fairness, Titan dynamically adapts the behavior of the many different layers of an OS's network stack to changes in network load and adds a new abstraction for programming the packet scheduler of a NIC. Specifically, Titan comprises the following components: Dynamic Queue Assignment (DQA), Dynamic Queue Weight Assignment (DQWA), and Dynamic Segmentation Offload Sizing (DSOS).

Given a fixed number of NIC queues, we target the three behavior modes of behavior we previously described: undersubscribed, low oversubscription, and high oversubcription. Titan is designed to improve server-side networking performance regardless of which mode a server currently is operating in, and the different components of Titan are targeted for improving performance in each of these different regimes. The rest of this section discusses the design of these components.

3.1 Dynamic Queue Assignment (DQA)

When it is possible for a segment to be placed in more than one queue, the OS must implement a queue assignment algorithm. In Linux, a per-socket hash is used to assign segments to queues. Even when there are fewer flows than queues (undersubscribed), hash collisions can lead to unfairness.

Titan uses Dynamic Queue Assignment (DQA) to avoid the problems caused by hash collisions when there are fewer flows than queues. Instead of hashing, DQA chooses the queue for a flow dynamically based on the current state of the software and hardware queues. DQA assigns flows to queues based on queue weights that are internally computed by Titan. In other words, there are

two components to DQA: an algorithm for computing the OS's internal weight for each queue and an algorithm for assigning a segment to a queue based on the current weight of every software/hardware queue that the segment can use.

Queue weight computation: Titan uses the current traffic that is enqueued in a software/hardware queue pair to compute a weight for each queue. We assume that the OS can assign a weight to each network flow based on some high-level policy. Titan dynamically tracks the sum of the weights of the flows sharing the same queue: it updates a queue's weight when a flow is first assigned to a queue and when a TX interrupt frees the last outstanding `skbuff` for the flow.

Queue assignment algorithm: Dynamically tracking queue occupancy can allow a queue assignment algorithm to avoid hash collisions. Our goals in the design of a DQA are to avoid packet reordering and provide accurate assignment without incurring excessive CPU utilization overheads. We use a greedy algorithm to assign flows to queues with the aim of spreading weight evenly across all queues. This algorithm selects the queue with the minimum weight.

The main overhead of our current implementation of DQA is that it reads the weights of every queue a flow may use. XPS reduces this overhead by reducing the number of queue weights that need to be read: if a flow is not allowed to use a queue, DQA will not read its weight. Although not necessary, our current implementation introduces a lock to serialize queue assignment per XPS pool. We are currently investigating using a lock-free priority queue to allow multiple cores to simultaneously perform queue assignment without reading every queue's weight while still avoiding choosing the same queues.

In order to avoid packet reordering, DQA only changes a flow's queue assignment when it has no outstanding bytes enqueued in a software or hardware queue. This also has the added benefit of reducing the CPU overheads of queue assignment because it will be run at most once per TX interrupt/NAPI polling interval and often only once for as long as a flow has backlogged data and is allowed to send by TCP. However, this also implies that unfairness can arise as flows complete because remaining flows are not rebalanced.

3.2 Dynamic Queue Weight Assignment (DQWA)

DQA computes queue weights to perform queue assignment. However, these queue weights are only an OS construct. The NIC does not perform scheduling decisions based on these weights; it services queues based on simple round-robin instead. During periods of oversubscription, this can lead to unfairness.

To solve this problem, Titan modifies NIC drivers to expose a queue weight abstraction whereby higher levels of the network stack can cause the NIC scheduler to service queues in proportion to the OS' weights. This is accomplished by introducing the new `ndo_set_tx_weight` network device operation (NDO) for drivers to implement. The OS calls this function whenever it updates a queue's weight, which allows the NIC driver to dynamically program the NIC scheduler. We call this Dynamic Queue Weight Assignment (DQWA). Although simple, this new function allows the NIC to generate a fair packet schedule provided that the NIC scheduler is capable of being programmed.

The main overhead of DQWA is that each update generates a PCIe write. Like DQA, DQWA weights only need to be changed at most once per TX interrupt/NAPI polling interval. However, if necessary, the number of DQWA updates can also be rate limited.

While not all commodity NICs allow weight setting, it is a small addition to mechanisms already present. A NIC scheduler must implement a scheduling algorithm that provides per-queue fairness even if different sized segments are enqueued. To modify this algorithm to service queues in proportion to different weights is simple; we borrow the classic networking idea of Deficit Round Robin (DRR) scheduling [36]. Specifically, by allocating each queue its own pool of credits that are decreased proportional to the number of bytes sent by the queue, DRR can provide per-queue fairness. Providing an interface to modify the allocation of credits to queues enables the NIC to configure DRR to service queues in proportion to different weights.

We implement the `ndo_set_tx_weight` in the `ixgbe` driver by configuring the NIC scheduler's per-queue DRR credit allocation.

3.3 Dynamic Segmentation Offload Sizing (DSOS)

When segments from competing flows share the same software/hardware queue pair, the size of a GSO segment becomes the minimum unit of fairness. Under periods of heavy oversubscription, the GSO size can become the major limiting factor on fairness because of the HOL blocking problems that large segments cause. Importantly, improving the interleaving of traffic from multiple different flows at finer granularities can also benefit network performance [18].

Currently, the only way to improve the fairness of software scheduling is by reducing the GSO size. However, this only improves fairness when multiple flows share a single queue. Otherwise, TSO interleaving in the NIC provides per-packet fairness independent of the

GSO (TSO) size. Reducing the GSO size when the network queues are not oversubscribed only wastes CPU.

Dynamic Segmentation Offload Sizing (DSOS) enables an OS to reduce GSO sizes for improved fairness under heavy load while avoiding the costs of reducing GSO sizes when NIC queues are not oversubscribed. This provides a better CPU utilization trade-off than was previously available.

In DSOS, packets are segmented from the default GSO size to a smaller segment size before being enqueued in the per-queue software queues only if multiple flows are sharing the same queue. (In our current implementation, re-segmentation happens in all queues as soon as there is oversubscription.) Segmentation in DSOS is identical to the implementation of GSO except that segmentation happens before `Qdisc` instead of after. Because the software queue (`Qdisc`) is responsible for fairly scheduling traffic from different flows, this enables the OS to generate a fair packet schedule while still benefiting from using large segments in the TCP/IP stack. Further, many multiqueue NICs also support passing a single segment as a scatter/gather list of multiple regions in memory. This enables a single large segment to be converted into multiple smaller segments without copying the payload data. If automatic TSO sizing generates segments smaller than the DSOS segment size, then no additional work is done.

4 Implementation

We implemented Titan in Linux 4.4.6 and modified Intel's out-of-tree `ixgbe-4.4.6` release [4] to support the new `ndo_set_tx_weight` NDO. We were able to implement this new NDO in this driver from the public hardware datasheets [2]. In a similar spirit, Titan is open source and available at https://github.com/bestephe/titan.

There is one major limitation in our current `ixgbe` driver implementation. We were only able to program the packet scheduler on the Intel 82599 NIC when it was configured in VMDq mode. As a side-effect, this causes the NIC to hash received packets (received side steering, or RSS) to only four RX queues. This effectively decreases the NIC's RX buffering capacity, so enabling this configuration can increase the number of packet drops. To try to mitigate the impact of reducing the receive buffering capacity of the NIC, we modified the `ixgbe-4.4.6` driver to enable a feature of the 82599 NIC that immediately triggers an interrupt when the number of available RX descriptors drops below a threshold.

During development, we found a problem with the standard Linux software queue scheduler. Linux tries to dequeue packets from software queues in a batch and enqueue them in their corresponding hardware queue whenever a segment is sent from any TCP flow. When multiple ACKs are received in a single interrupt, multiple TCP flows may try to create new `skbuff`s and enqueue them. If no bytes are enqueued in the software queues for two flows, and then ACKs for both flows arrive, the second flow will not have a chance to enqueue new `skbuff`s in the software queues before packets are dequeued from the software queue until the hardware queue is filled up to the BQL limit. In general, sending segments to the NIC as soon as the first TCP flow sends a segment may cause later TCP flows to miss an opportunity to send, leading to unfairness.

In Titan, we improve fairness with *TCP Xmit Batching*. With this mechanism, all of the TCP flows that enqueue segments at the same time in TSQ are allowed to enqueue packets into their respective software queues before *any* packets are dequeued from software queues and enqueued in the hardware queues. This is accomplished by changing the per-CPU TSQ tasklet in Linux so enqueuing a segment returns a pointer to a `Qdisc`. Packets are dequeued from the returned `Qdisc`s only after all pending segments have been enqueued.

5 Methodology

To evaluate Titan, we perform experiments by sending data between two servers and within a cluster of servers.

In the two server experiments, we use a cluster of three servers connected to a dedicated TOR switch via 10 Gbps Ethernet cables. One server is a source, another a sink, and the third server is for monitoring. The switch is a Broadcom BCM956846K-02. The first and second server are the traffic source and sink respectively. Both of these servers have a 4-core/8-thread Intel Xeon E5-1410 CPU, 24GB of memory, and connect to the TOR with Intel 82599 10 Gbps NICs [2]. We configure the switch to use port mirroring to direct all traffic sent by the first server to the third server. To monitor traffic, this server uses an Intel NetEffect NE020 NIC [5], which provides packet timestamps accurate to the microsecond.

We perform two types of two server experiments. First, we generate traffic using at most one `iperf3` [3] client per core pinned to different CPUs. Each client only uses a single thread. Because the fairness problems only arise when load is asymmetric, we distribute the flows across cores such that half of the cores have twice as many active flows as the other half of the cores. To measure latency, we use `sockperf` [8]. To measure CPU utilization, we use `dstat`. To avoid impacting CPU utilization by measuring latency, we measure latency and CPU utilization in separate experiments. Second, we use YCSB [12] to request both small and large values from memcached from different threads. We perform all of

the two server experiments with the NIC configured in VMDq mode.

In the cluster workloads, we use a cluster of 24 servers on CloudLab. Each of the servers has 2 10-core Intel E5-2660 v2 CPUs and 256GB of memory. All the servers connect to a Dell Networking S6000 switch via Intel 82599 NICs. Inspired by shuffle workloads used in prior work [13, 33, 22], we have all 24 servers simultaneously open a connection to every other server and send 1GB. We measure flow completion times. Because `iperf3` opens up additional control connections that can impact performance, we use a custom application to transfer data in this workload.

We compare Titan against two base configurations: Cvanilla, which is the default Linux configuration, and Cmax, which uses the MQ configuration system with a GSO size of 64KB, a TCP small queues limit of 256KB, and byte queue limits manually set to 256KB. In Cmax, interrupt coalescing on the NIC is also configured so that the NIC will use an interrupt interval of 50μs. In other words, the NIC will wait at least 50μs after raising an interrupt before it will be raised again. In the 2 server experiments, the traffic sink always uses configuration Cmax. Large receive offload (LRO) is disabled in all of the experiments because it can increase latency. We perform all experiments 10 times and report the average.

6 Evaluation

First, we evaluate the performance impact of individual components of Titan in the absence of any network congestion. Second, we evaluate Titan on a cluster of servers. In summary, we find that Titan is able to improve fairness on multiqueue NICs while only having a small impact on other metrics.

We study the following four metrics:

1. We measure *CPU utilization* as the sum percent of the time each core was not idle during a one second interval, summed across all cores and averaged across the duration of the experiment.
2. We measure *network throughput* as the total number of bytes that were sent per second across all flows, averaged across the duration of the experiment.
3. We measure *latency* with `sockperf` and report average latency. When we configure Linux software queues (`Qdiscs`), we prioritize the port used by `sockperf` above all other traffic.
4. We use a *normalized fairness metric* inspired by Shreedhar and Varghese [36]. For every flow $i \in F$, there is some configurable quantity f_i that expresses i's fair share. In all of our experiments, f_i is 1. If $sent_i(t_1,t_2)$ is the total number of bytes sent by flow i in the interval (t_1,t_2), then the fairness metric FM is as follows:

$$FM(t_1,t_2) = max\{i,j \in F | sent_i(t_1,t_2)/f_i - sent_j(t_1,t_2)/f_j\}$$

In other words, the fairness metric $FM(t_1,t_2)$ is the instantaneous *worst case difference* in the normalized bytes sent by any two competing flows over the time interval. Ideally, the fairness metric should be a small constant no matter the size of the time interval [36].

For our experiments, we do not report this ideal FM but instead use normalized fairness *NFM(τ)*, which is the fairness metric FM over all intervals of duration τ, normalized to the fair share of data for a flow in the interval.

$$NFM(\tau) = FM(\tau) * \frac{line_rate * \tau^{-1}}{\sum_{j \in F} f_j}$$

For example, with 10 flows, a flow's fair share of a 10 Gbps link over 1 second is 128MB; if the highest FM over a 1-second interval is 64 MB, then NFM is 0.5. Note that NFM can exceed 1 when some flows get much higher performance than others.

6.1 Two Server Performance

There are multiple complementary components to Titan, and we evaluate the impact of individual components on performance in the absence of network congestion. Table 3 shows the performance of different components of Titan for each metric. The expected benefit of Titan is improved fairness, but it is possible for Titan to hurt throughput, latency, or CPU utilization. These results show that Titan is able to significantly improve fairness often without hurting throughput and latency and with a small increase in CPU utilization (often $< 10\%$)

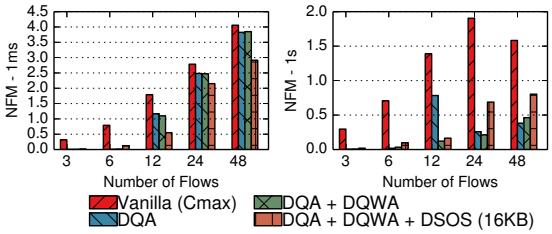
Dynamic Queue Assignment: DQA ensures that when there are fewer flows than queues, each flow is assigned its own queue. The Cmax (hashing) and DQA results in Figure 2 shows the fairness differences between using hashing and DQA for assigning flows to queues given 8 hardware queues and a variable number of flows. We report *NFM*, the normalized fairness metric.

With hashing, fairness is good with 3 flows as there are few collisions. However, with more flows, the unfairness of hashing is high at short and long timescales because there are often hash collisions. Unfairness is bad because of HOL blocking while waiting for GSO/TSO-size segments and hashing leading to uneven numbers of flows per queue.

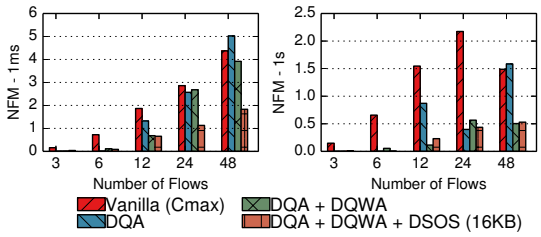
In contrast, with DQA there is no unfairness in the undersubscribed case, as DQA always assigns every flow its own queue. In the low oversubscription case of 12 flows, there is also unfairness because some flows must

Config	3 flows, 8 Queue (1 per CPU) TPut (Gbps)	CPU (%)	Latency (μs)	NFM (1ms)	NFM (1s)	Config	12 flows, 8 Queues (1-per CPU) TPut (Gbps)	CPU (%)	Latency (μs)	NFM (1ms)	NFM (1s)
Cvanilla	9.4	64	298	0.33	0.31	Cv:	9.4	58	912	1.83	1.23
Cmax: SFQ/Prio + 256KB BQL	9.4	72	125	0.16	0.15	Cmax:	9.4	55	912	1.79	1.39
Titan1: DQA	9.4	78	123	0.00	0.00	T1:	9.4	66	657	1.17	0.78
Titan2: DQA + DQWA	9.4	77	124	0.00	0.01	T2:	9.4	70	516	1.10	0.12
Titan3: DQA + DQWA + DSOS (16KB)	9.4	82	180	0.02	0.02	T3:	9.4	96	395	0.55	0.17
XPS: Cmax + XPS	9.4	54	130	0.16	0.15	XPS:	9.4	55	526	1.87	1.55
TitanXPS1: DQA	9.4	55	126	0.00	0.00	TX1:	9.4	49	660	1.32	0.87
TitanXPS2: DQA + DQWA	9.4	57	121	0.01	0.00	TX2:	9.4	50	505	0.68	0.11
TitanXPS3: DQA + DQWA + DSOS (16KB)	9.4	65	128	0.04	0.01	TX3:	9.4	59	269	0.66	0.23
	48 flows, 8 Queue (1 per CPU)						192 flows, 8 Queues (1-per CPU)				
Cvanilla	9.4	72	2019	5.01	1.95	Cv:	9.4	98	3881	15	3.32
Cmax: SFQ/Prio + 256KB BQL	9.4	83	653	4.06	1.58	Cmax:	9.1	109	604	6.93	1.39
Titan1: DQA	9.4	89	660	3.83	0.38	T1:	9.5	118	554	8.35	0.54
Titan2: DQA + DQWA	9.4	87	585	3.85	0.46	T2:	9.5	103	509	8.42	0.49
Titan3: DQA + DQWA + DSOS (16KB)	9.3	103	285	2.92	0.80	T3:	9.4	113	342	3.50	0.80
XPS: Cmax + XPS	9.4	53	639	4.37	1.49	XPS:	9.5	119	517	10	2.66
TitanXPS1: DQA	9.4	61	660	5.02	1.58	TX1:	9.5	113	552	8.46	0.76
TitanXPS2: DQA + DQWA	9.4	62	606	3.92	0.50	TX2:	9.5	123	519	8.28	0.57
TitanXPS3: DQA + DQWA + DSOS (16KB)	9.4	76	333	1.83	0.53	TX3:	9.4	138	300	3.50	0.81

Table 3: The performance of different OS configurations given 3, 12, 48, and 192 flows spread across 8 cores.



(a) No XPS



(b) XPS

Figure 2: The impact of the individual aspects of Titan on short-term and long-term fairness.

share queues, and without DQWA to program weights in the NIC, all queues are serviced equally. With 48 flows, DQA has low unfairness over long timescales because it will place exactly 6 flows in each queue.

Dynamic Queue Weight Assignment: DQWA enables an OS to pass queue weights, in this case the number of flows, to the NIC so that queues with more flows receive more service. Figure 2 shows the fairness of the DQA queue assignment algorithms when DQWA is enabled. These results show that over short timescales, DQWA has little impact as it takes time for queue weights to fix transient unfairness, and in highly oversubscribed cases HOL blocking is the major cause of unfairness. Over longer timescales, DQWA improves the fairness at low levels of oversubscription because the NIC is able to give more service to queues with more flows. At high levels of oversubscription, DQA is able to evenly distribute flow weights across queues, so DQWA is not able to further improve fairness.

We note that DQA is a software-only solution that has the largest impact in undersubcribed cases and helps at both short and long timescales. DQWA helps most in (i) oversubscribed cases and (ii) over longer timescales. In addition, DQWA requires hardware support that, while minimal, may not be present in all NICs. Also, we evaluated DQWA with hashing instead of DQA, and we found that DQWA also improves fairness without DQA.

Dynamic Segmentation Offload Sizing: DSOS addresses HOL blocking by reducing segment size from the default 64KB to a smaller size dynamically under oversubscription. We compare DQA and DQWA with and without DSOS for 16KB DSOS segment sizes. Figure 2 shows that DSOS improves fairness at the 1ms timescale. In the 3 and 6 flow cases there is no oversubscription, so DSOS leaves the GSO size at 64KB. For 12, 24, and 48 flows, though, DSOS reduces the segment size to reduce HOL blocking. At short timescales, this improves fairness. Over longer timescales, DSOS can slightly hurt fairness. This is because DSOS can increase CPU utilization.

XPS: So far, our evaluation has focused our discussion on the multiqueue NIC configuration (MQ). Transmit packet steering (XPS; Section 2.1) assigns pools of queues to pools of CPUs and behaves differently than MQ. To understand these differences, Figure 2 also shows the fairness of Titan when XPS is configured. For the most part, this figure shows that XPS has little impact

on network fairness in Titan.

The biggest change in Figure 2 is that XPS improves the fairness of DSOS (with both DQA and DQWA enabled) at short timescales during oversubscription. When there are 48 flows, using a 16KB dynamic segment size with XPS almost halves NFM at short time scales. The reason for this is because XPS reduces the CPU overheads of DSOS (Table 3). This is because XPS improves cache locality.

CPU Utilization, Throughput and Latency: While the goal of Titan is improved fairness, it must not come at the cost of increased CPU utilization, decreased throughput, or increased latency. Tables 3 compares the performance of Titan with Cvanilla and Cmax.

At all subscription levels, throughput is almost always identical with Titan and standard Linux networking options. Similarly, CPU utilization is slightly higher with Titan. It must do more work for queue assignment and weight-setting. During oversubscription, DSOS must segment and process smaller segments. Fortunately, enabling XPS reduces the CPU utilization of all of the features of Titan.

Regardless of the subscription level, Titan can increase latency. In the absence of any other traffic, the average baseline latency we observed is 32μs. In the presence of bulk transfers, the minimum average latency we observe is 121μs, and the highest average latency we observe is 3.9ms. This high latency is because the HOL blocking latency of the NIC (for a given priority) is at least equal to the minimum number of bytes enqueued in any queue multiplied by the number of active queues. Although we find that latency in general is high, we observe that Titan does not significantly hurt latency. The latency of Titan is often near that of Cmax, and at most Titan increases latency by 134μs. When NIC queues are oversubscribed, we observe that DSOS can reduce latency by over 200μs. Further, we also looked at tail latency and found that the 90th percentile latency for Titan is never more than 200μs higher than the average.

Currently, the best practice for addressing this problem is to use DCB priorities to isolate high priority traffic onto independent pools of NIC queues that are serviced with higher priority by the NIC hardware. Traffic in one DCB priority is not able to increase the latency of traffic in a higher DCB priority.

In summary, we find that overall Titan greatly improves fairness across a wide range of subscription levels, often at no or negligible throughput or latency overheads. Titan can cause a small increase in CPU utilization, often less than 10%. At most, this increase is 17% and 27% with and without XPS, respectively.

Finally, we have also performed experiments to evaluate the impact of Titan on average and tail request completion times in memcached. These experiments use

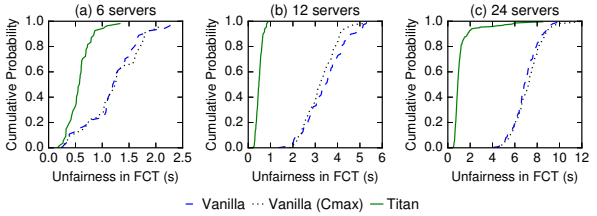


Figure 3: The impact of Titan on fairness on a cluster of servers performing a shuffle.

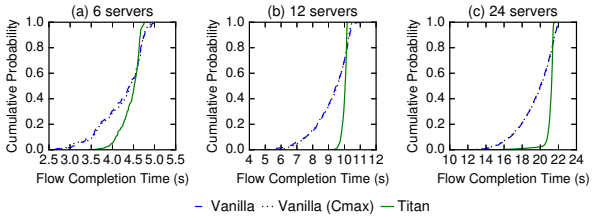


Figure 4: The impact of Titan on flow completion times (FCT) on a cluster of servers performing a shuffle.

YCSB with 7 request threads, 6 of which request 512KB values, while the remaining thread requests small objects (2–64KB). We find that Titan is able to reduce the average and 99th percentile completion times for the small objects by 3.2–10.6% and 7.3–32%, respectively. This is because Titan is able to avoid HOL blocking latency through dynamic queue assignment.

6.2 Cluster Performance

In order to evaluate the cluster performance of Titan, we measure the impact of improving the fairness of the packet stream emitted by a server when there is network congestion and when there are more communicating servers. To do so, we perform an all-to-all shuffle for different cluster sizes where each server simultaneously opens connections to every other server and transfers 1GB of data. This workload is inspired by the shuffle phase of Map/Reduce jobs.

Figure 3 shows the impact of Titan on network performance in a cluster of 6, 12, and 24 servers. We plot a CDF of the difference in the completion time of the earliest completing flow and that of the last completing flow. First, Figure 3 confirms that without Titan flow fairness is a problem in a cluster of servers. Both the default Linux configuration (Cvanilla) and an optimized Linux configuration (Cmax) behave similarly and show substantial variation in completion times. In contrast, with Titan unfairness substantially improves at all three subscription levels and is consistently much better than Cvanilla and Cmax.

Further, we find that Titan is not only able to improve fairness, but that improving fairness also reduces the tail

flow completion times (>80th percentile) for the flows in the shuffle as well. To show why, Figure 4 shows a CDF of the flow completion times across all the flows in the shuffle for different cluster sizes. This figure shows that Titan provides more consistent flow completion times. Because of this, the fastest flows (<20th percentile) in Cvanilla and Cmax complete faster. However, this comes at the expense of tail flow completion times. Figure 4 shows that Titan can reduce the tail of the flow completion time distribution (>80th percentile).

Finally, for this test, DQA (without DQWA or DSOS) is enough to get most of the fairness benefit of Titan. At small cluster sizes, we found that DQWA can still further improve fairness. Unfortunately, we discovered that configuring our NICs into VMDq mode reduces RX buffering capacity and hurts completion times. Because our implementation of DQWA requires VMDq mode to program queue weights, we cannot evaluate DQWA's benefit for large clusters.

7 Related Work

Titan is closely related to SENIC [31] and Silo [23][1]. SENIC argues that NICs in the future will be able to provide enough queues such that two flows will never have to share the same queue. In contrast, Silo builds a system for fairly scheduling traffic from competing VMs using a single transmit queue (SQ) because of the control it gives to the OS. Titan introduces a middle ground that can achieve some of the benefits of both designs.

Many projects in addition to Silo have used the SQ model. In particular, the SQ model is popular for emulating new hardware features not yet provided by the underlying hardware [31, 21, 25]. This is because it provides the OS with the most control over packet scheduling.

Similar to Titan, PSPAT [34] performs per-packet scheduling in a dedicated kernel thread that is separated from applications and device drivers with two sets of lock-free queues. Making per-packet scheduling decisions in PSPAT instead of per-segment decisions in Titan can significantly improve fairness and latency, and Titan can cause PCIe contention that is avoided in PSPAT by only issuing PCIe writes from a single core. If PSPAT were extended to use multiple independent scheduling threads to drive independent NIC queues, then programming the NIC scheduler with DQWA in Titan would be complementary.

There has been recent work on building networks that provide programmable packet scheduling [38, 29, 16], allowing flows to fairly compete [15, 41, 39], and performing traffic engineering in the network [13, 22, 17, 33, 14, 18]. Titan is motivated by similar concerns and is complementary. If the packet schedule emitted by a server is not fair, then the end-server can become the main limiting factor on fairness, not the network. Thus, Titan can improve the efficacy of the aforementioned techniques.

Affinity-Accept [30] improves connection locality on multicore processors, and Fastsocket [27] improves the multicore scalability of the Linux stack when a server handles many short-lived network connections. Titan is complementary to both of these designs. Titan benefits from their improvements in connection setup, while these designs can benefit from improved flow fairness in Titan.

[1]The Titan Missile Museum is located in a silo. We imagine it is scenic.

8 Conclusions

With increasing datacenter (DC) server line rates it becomes important to understand how best to ensure that DC applications can saturate high speed links, while also ensuring low latency, low CPU utilization, and per-flow fairness. While modern NICs and OS's support a variety of interesting features, it is unclear how best to use them towards meeting these goals. Using an extensive measurement study, we find that certain multi-queue NIC configurations are crucial to ensuring good latency, throughput and CPU utilization, but substantial unfairness remains. To this end, we designed Titan, an extension to the Linux network stack that incorporates three main ideas – dynamic queue assignment, dynamic queue weights, and dynamic segmentation resizing. Our evaluation using both experiments between two servers on an uncongested network and between a cluster of servers shows that Titan can reduce unfairness across a range of conditions while minimally impacting the other metrics.

Titan is complementary with a variety of other DC host networking optimizations, such as DCB and receive-side network optimizations. Titan's sender-side fairness guarantees are crucial to ensure the efficacy of in-network fair-sharing mechanisms. Finally, the three main ideas in Titan can be employed alongside other systems, e.g., those for DC-wide traffic scheduling and other existing systems optimized for short-lived connections.

9 Acknowledgements

We would like to thank our shepherd Michio Honda and the anonymous reviewers for their help and insightful feedback. This work is supported by the National Science Foundation grants CNS-1654843 and CNS-1551745.

References

[1] bql: Byte queue limits. https://lwn.net/Articles/469652/.

[2] Intel 82599 10 GbE controller datasheet. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf.

[3] iperf3: Documentation. http://software.es.net/iperf/.

[4] ixgbe-4.4.6. https://sourceforge.net/projects/e1000/files/ixgbe%20stable/4.4.6/.

[5] Neteffect server cluster adapters. http://www.intel.com/content/dam/doc/product-brief/neteffect-server-cluster-adapter-brief.pdf.

[6] rfs: Receive flow steering. http://lwn.net/Articles/381955/.

[7] rps: Receive packet steering. http://lwn.net/Articles/361440/.

[8] sockperf: Network benchmarking utility. https://github.com/Mellanox/sockperf.

[9] tcp: TSO packets automatic sizing. https://lwn.net/Articles/564979/.

[10] tsq: Tcp small queues. https://lwn.net/Articles/506237/.

[11] xps: Transmit packet steering. https://lwn.net/Articles/412062/.

[12] Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB/wiki.

[13] AL-FARES, M., RADHAKRISHNAN, S., RAGHAVAN, B., HUANG, N., AND VAHDAT, A. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (2010), NSDI '10.

[14] ALIZADEH, M., EDSALL, T., DHARMAPURIKAR, S., VAIDYANATHAN, R., CHU, K., FINGERHUT, A., LAM, V. T., MATUS, F., PAN, R., YADAV, N., AND VARGHESE, G. CONGA: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM Conference on Special Interest Group on Data Communication* (2014), SIGCOMM '14.

[15] 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 2010 ACM Conference on Special Interest Group on Data Communication* (2010), SIGCOMM '10.

[16] ALIZADEH, M., YANG, S., SHARIF, M., KATTI, S., MCKEOWN, N., PRABHAKAR, B., AND SHENKER, S. pFabric: Minimal near-optimal datacenter transport. In *Proceedings of the 2013 ACM Conference on Special Interest Group on Data Communication* (2013), SIGCOMM '13.

[17] BENSON, T., ANAND, A., AKELLA, A., AND ZHANG, M. MicroTE: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh ACM Conference on Emerging Networking Experiments and Technologies* (2011), CoNEXT '11.

[18] GENG, Y., JEYAKUMAR, V., KABBANI, A., AND ALIZADEH, M. Juggler: A practical reordering resilient network stack for datacenters. In *Proceedings of the Eleventh ACM European Conference on Computer Systems* (2016), EuroSys '16.

[19] GETTYS, J., AND NICHOLS, K. Bufferbloat: Dark buffers in the internet. *Queue 9*, 11 (Nov. 2011), 40:40–40:54.

[20] 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 2015 ACM Conference on Special Interest Group on Data Communication* (2015), SIGCOMM '15.

[21] 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.

[22] HE, K., ROZNER, E., AGARWAL, K., FELTER, W., CARTER, J. B., AND AKELLA, A. Presto: Edge-based load balancing for fast datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), SIGCOMM '15.

[23] JANG, K., SHERRY, J., BALLANI, H., AND MONCASTER, T. Silo: Predictable message latency in the cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), SIGCOMM '15.

[24] KAPOOR, R., SNOEREN, A. C., VOELKER, G. M., , AND PORTER, G. Bullet trains: a study of NIC burst behavior at microsecond timescales. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies* (2013), CoNEXT '13.

[25] KAUFMANN, A., PETER, S., SHARMA, N. K., ANDERSON, T., AND KRISHNAMURTHY, A. High performance packet processing with FlexNIC. In *Proceedings of the Twenty-First ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2016), ASPLOS '16.

[26] KUMAR, A., JAIN, S., NAIK, U., KASINADHUNI, N., ZERMENO, E. C., GUNN, C. S., AI, J., CARLIN, B., AMARANDEI-STAVILA, M., ROBIN, M., SIGANPORIA, A., STUART, S., AND VAHDAT, A. BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), SIGCOMM '15.

[27] LIN, X., CHEN, Y., LI, X., MAO, J., HE, J., XU, W., AND SHI, Y. Scalable kernel TCP design and implementation for short-lived connections. In *Proceedings of the Twenty-First ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2016), ASPLOS '16.

[28] MENON, A., AND ZWAENEPOEL, W. Optimizing TCP receive performance. In *Proceedings of the 2008 USENIX Conference on Annual Technical Conference* (2008), USENIX ATC '08.

[29] MITTAL, R., AGARWAL, R., RATNASAMY, S., AND SHENKER, S. Universal packet scheduling. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks* (2015), HotNets-XIV.

[30] PESTEREV, A., STRAUSS, J., ZELDOVICH, N., AND MORRIS, R. T. Improving network connection locality on multicore systems. In *Proceedings of the Seventh ACM European Conference on Computer Systems* (2012), EuroSys '12.

[31] RADHAKRISHNAN, S., GENG, Y., JEYAKUMAR, V., KABBANI, A., PORTER, G., AND VAHDAT, A. SENIC: Scalable NIC for end-host rate limiting. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), NSDI '14.

[32] RAM, K. K., COX, A. L., CHADHA, M., AND RIXNER, S. Hyper-switch: A scalable software virtual switching architecture. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (2013), USENIX ATC '13.

[33] 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 *Proceedings of the 2014 ACM Conference on Special Interest Group on Data Communication* (2014), SIGCOMM '14.

[34] RIZZO, L., VALENTE, P., LETTIERI, G., AND MAFFIONE, V. PSPAT: Software packet scheduling at hardware speed. http://info.iet.unipi.it/~luigi/pspat/. Preprint; accessed May 31 2017.

[35] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), SIGCOMM '15.

[36] SHREEDHAR, M., AND VARGHESE, G. Efficient fair queueing using deficit round robin. In *Proceedings of the 1995 ACM Conference on Special Interest Group on Data Communication* (1995), SIGCOMM '95.

[37] 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. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), SIGCOMM '15.

[38] SIVARAMAN, A., SUBRAMANIAN, S., ALIZADEH, M., CHOLE, S., CHUANG, S.-T., AGRAWAL, A., BALAKRISHNAN, H., EDSALL, T., KATTI, S., AND MCKEOWN, N. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication* (2016), SIGCOMM '16.

[39] STEPHENS, B., COX, A. L., SINGLA, A., CARTER, J. B., DIXON, C., AND FELTER, W. Practical DCB for improved data center networks. In *Proceedings of the 33rd Annual IEEE International Conference on Computer Communications* (2014), INFOCOM '14.

[40] STEPHENS, B., SINGHVI, A., AKELLA, A., AND SWIFT, M. Titan: Fair packet scheduling for commodity multiqueue NICs. Tech. Rep. TR1840, University of Winconsin-Madison, Department of Computer Sciences, February 2017. http://digital.library.wisc.edu/1793/75739.

[41] VAMANAN, B., HASAN, J., AND VIJAYKUMAR, T. Deadline-aware datacenter tcp (D2TCP). In *Proceedings of the 2012 ACM Conference on Special Interest Group on Data Communication* (2012), SIGCOMM '12.

MopEye: Opportunistic Monitoring of Per-app Mobile Network Performance

Daoyuan Wu[1*], Rocky K. C. Chang[2], Weichao Li[2], Eric K. T. Cheng[2], and Debin Gao[1]

[1]Singapore Management University
[2]The Hong Kong Polytechnic University

https://mopeye.github.io[†]

Abstract

Crowdsourcing mobile user's network performance has become an effective way of understanding and improving mobile network performance and user quality-of-experience. However, the current measurement method is still based on the landline measurement paradigm in which a measurement app measures the path to fixed (measurement or web) servers. In this work, we introduce a new paradigm of measuring *per-app* mobile network performance. We design and implement MopEye, an Android app to measure network round-trip delay for each app whenever there is app traffic. This opportunistic measurement can be conducted automatically without user intervention. Therefore, it can facilitate a large-scale and long-term crowdsourcing of mobile network performance. In the course of implementing MopEye, we have overcome a suite of challenges to make the continuous latency monitoring lightweight and accurate. We have deployed MopEye to Google Play for an IRB-approved crowdsourcing study in a period of ten months, which obtains over five million measurements from 6,266 Android apps on 2,351 smartphones. The analysis reveals a number of new findings on the per-app network performance and mobile DNS performance.

1 Introduction

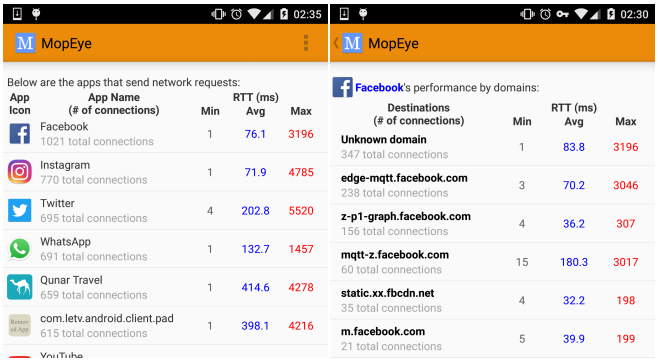
In recent years, a number of crowdsourcing platforms using smartphone apps are deployed to measure mobile network performance. MobiPerf [5] and Netalyzr [7] on Android, for example, enable users to measure a number of network performance metrics between their smartphones and remote endpoints. Using these uncoordinated network measurement performed by end users to obtain accurate and meaningful insights is still under active research [40]. Related to that, a number of speedtest services are provided for Android [13, 16], iOS [14, 15], and Windows Phone users [8, 17].

The existing mobile measurement apps, however, are still based on the landline measurement paradigm. They actively send probe packets to user-specified remote endpoints or measurement servers (e.g., M-Lab servers). Due to the diverse locations of various servers and user mobility, such landline measurement will not correlate well with the user's experience. In this paper, we propose to measure mobile network performance for each app (i.e., from user's smartphone to the app server). The *per-app* measurement not only reflects user's experience with the app but also helps diagnose application-specific problems. An effective approach to per-app measurement is to perform the measurement only when there is app traffic. Since this opportunistic measurement can be conducted automatically without user's intervention, it can facilitate a large-scale and long-term crowdsourcing of mobile network performance.

In this paper, we utilize the `VpnService` API available on Android 4.0+ [20] to implement opportunistic measurement of per-app network performance in MopEye (MObile Performance Eye), our Android measurement app. Figure 1 shows the two main interfaces in MopEye. With the `VpnService` interface, MopEye can *passively* capture the traffic initiated by all apps and forward them *actively* to the remote app servers using socket calls. Based on the `connect()` socket calls, it can estimate the round-trip time (RTT) for each app. Therefore, the measurement incurs zero network overhead, and the RTT can accurately reflect the network delay experienced by each app. Moreover, MopEye can be deployed easily, because it does not need the root privilege which is required for `tcpdump`-based passive measurement. It is also very easy to operate. Users are only required to

*Half of the work by this author was performed at The Hong Kong Polytechnic University.

[†]We thank Dr. Ada Gavrilovska for shepherding our paper and the anonymous reviewers for their valuable comments. This work is partially supported by a grant (ref. no. G-YBAK) from The Hong Kong Polytechnic University, a grant (ref. no. H-ZL17) from the Joint Universities Computer Centre of Hong Kong, and the Singapore National Research Foundation under NCR Award Number NRF2014NCR-NCR001-012.



(a) An all-app view. (b) An individual-app view.

Figure 1: MopEye's two major user interfaces.

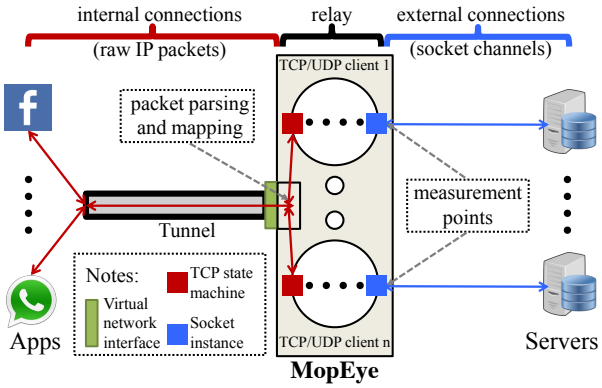


Figure 2: An overview of MopEye.

consent to enabling MopEye's VPN interface once. After that, MopEye performs the measurement opportunistically and autonomously.

The main challenge in the design and implementation of MopEye is to mitigate the impact on other apps by performing *fast* packet relaying. However, our design choices are constrained by two important restrictions: no relaying using a remote VPN server and no raw sockets which require the root privilege. To satisfy the constraints, we build our own user-space TCP/IP stack to perform packet relaying between the VPN tunnel packets and those in the socket connections. In particular, we have identified and overcome a number of serious performance degradation issues in the entire packet-relaying process. Another challenge is to obtain high measurement accuracy. Based on our evaluation, MopEye's mean RTT measurement deviates from `tcpdump`'s results by at most 1ms. Besides that, our evaluation also shows that MopEye incurs very low overhead on the throughput, battery consumption, and CPU usage.

We have deployed MopEye to Google Play [6] for an IRB-approved crowdsourcing study since May 2016. We have so far[1] attracted 4,014 user installs from 126 countries and collected the *first* large-scale per-app measurement dataset comprising 5,252,758 RTT measurements from 6,266 Android apps on 2,351 smartphones[2]. An analysis of these crowdsourced data reveals a number of new findings on the per-app and DNS network performance experienced by real users under different network types and ISPs in the wild. We also perform several case studies to diagnose the performance issues in Whatsapp, India's largest 4G ISP, and two American cellular ISPs.

2 Design of MopEye

In this section, we present an overview of MopEye and its main components. We defer the implementation details and performance enhancement to the next section.

[1]By the time of our submission on 7 February 2017.

[2]Note that many users use daily apps such as Facebook and Whatsapp. Thus, there is a large common app space among different phones.

2.1 MopEye Overview

Figure 2 presents a high-level design of MopEye. There are three main steps for MopEye to use an app's traffic to opportunistically measure the network RTT. For the outgoing traffic, MopEye first captures an app's packets through a tunnel, relays the captured packet to an external TCP connection or UDP association with a remote server, and sends the packets to the server. In the last step, MopEye calculates the time between the app's SYN and SYN/ACK packets to measure the RTT. The RTT measurement for UDP apps is similar (i.e., between query and response messages). In the following we describe each step in more details.

2.2 Packet Capturing, Parsing, and Mapping

We leverage Android's `VpnService` APIs to build a virtual network interface (green box in Figure 2) to intercept all traffic initiated from any app on the smartphone. It also receives server-initiated traffic, but for the sake of simplicity we do not discuss this traffic direction in this paper.

Android's `VpnService` APIs leverage the `TUN` virtual network device (`/dev/tun` on Android or `/dev/net/tun` on some UNIX systems) to capture packets. Figure 3 illustrates MopEye's packet capturing and relaying mechanisms for the incoming and outgoing traffic. Once MopEye builds a `TUN` interface (i.e., `mInterface` in the figure), the `TUN` device driver will capture and deliver all outgoing app packets to this interface. MopEye then obtains these packets using `mInterface`'s input stream. It is worth noting that the packets captured here are all IP packets, because a `TUN` device is essentially a virtual point-to-point IP link. MopEye parses the captured packets to obtain the IP addresses and port numbers for packet relaying.

To support per-app measurement, MopEye must also determine to which app a captured packet belongs. Although there is no API support for this socket-to-app mapping function, we find that four pseudo files in the

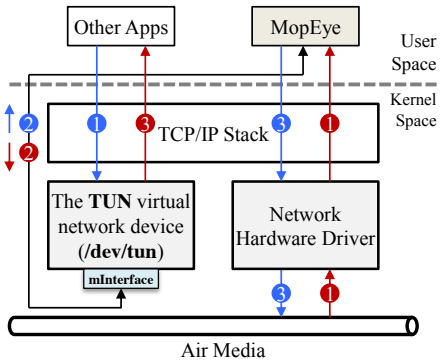


Figure 3: MopEye's packet capturing and relaying for incoming flow (red) and outgoing flow (blue). The black link represents a bi-directional flow.

`proc` filesystem (`/proc/net/tcp6|tcp|udp|udp6`) store each TCP/UDP connection's local and remote IP addresses and ports, as well as the corresponding app's UID which is a unique ID for each installed app. Moreover, using Android's `PackageManager` APIs, MopEye obtains the app's name from its UID. To reduce the overhead of this procedure, MopEye performs this operation only for the SYN packets, and the resolved names and socket addresses are cached for the subsequent data packets. Furthermore, we will present in §3.3 a new mechanism to significantly minimize the mapping overhead even for SYN packets. As for UDP packets, MopEye currently supports only DNS measurement (though it relays all UDP packets). Since DNS is system-wide, MopEye does not need to map UDP packets for now.

2.3 Packet Relaying

Relaying packets between apps and their servers efficiently is the most challenging task in the design and implementation of MopEye. Our solution to this problem is shaped by the three main considerations below.

- **Measurement objective** Since our goal is to measure the RTT between a user's smartphone and the app servers, we cannot rely on a remote VPN server to relay the application packets to their servers. Therefore, we require MopEye to relay packets within the smartphone.
- **Running on unrooted phones** Our another objective is to run MopEye on unrooted phones. Using raw sockets to relay packets to the servers is therefore not an option. Instead, MopEye must relay packets via the regular TCP/UDP sockets for the external connections. We have implemented both TCP and UDP packet relays. Due to the page limit, we describe only the TCP relay from now on.
- **User-space TCP stack** As a result of using regular TCP socket, MopEye will not be able to access the information in the TCB (Transmission Control Block [11]), such as the TCP sequence and acknowledgement numbers, from the external connections. Therefore, MopEye must create its own user-space TCP stack (in the form of TCP state machine) for the internal connections. We refer the packets transmitted in the internal and external connections to as *tunnel packets* and *socket packets*, respectively.

Splicing the two connections To relay packets in a TCP connection, MopEye "splices" the internal connection terminated by MopEye's TCP state machine and the external connection initiated by MopEye's TCP socket. Our approach is to link the state machine and the socket with two-way referencing. That is, we create a TCP client object that wraps the socket instance and include a reference to the state machine. The state machine also maintains a reference to the corresponding TCP client.

Processing tunnel packets MopEye processes the tunnel packets according to RFC 793 [11]. The processing logics for different TCP packets are summarized as follows.

- *TCP SYN:* Upon receiving a SYN packet, MopEye creates a TCP client object and uses its socket instance to perform handshake with the remote server. Only after establishing the external connection can MopEye complete the handshake with the app.
- *TCP Data:* MopEye places the data from tunnel packets to a socket write buffer and triggers a socket write event for the socket instance to handle.
- *Pure ACK:* MopEye discards pure ACK packets, because there is no need to relay them to the socket channel.
- *TCP FIN:* MopEye updates the TCP state to half closed and generates an ACK packet to the app. Meanwhile, it triggers a half-close write event for the socket instance to handle.
- *TCP RST:* MopEye closes the external socket connection and removes the corresponding TCP client object from the cached TCP client list.

Processing socket packets To handle concurrent socket instances, MopEye uses *non-blocking* `SocketChannel` APIs to communicate with the remote app servers. In particular, it uses a socket selector [32] to listen for read and write events, and handles them as follows.

- *Socket Read:* Upon detecting a read event, MopEye retrieves the incoming data from the read buffer and constructs data packets for the internal connection. In §3.4, we propose a method to improve the performance of this step. However, if this read event is for a socket close/reset, MopEye generates a FIN/RESET packet for the internal connection.
- *Socket Write:* Upon detecting a write event, the socket instance sends all the data in the write buffer to the remote server and instructs the corresponding TCP state machine to generate an ACK packet to the app. However, if this write event is for half-close, MopEye closes the external connection and generates a FIN packet to the app.

2.4 Measurement Methods

Obtaining accurate per-app RTT measurement using MopEye faces more challenges than that using the traditional active measurement apps, such as MobiPerf [5] and Ookla Speedtest [16]. There are two main challenges.

C1: Since MopEye has no control on the relayed packets, it cannot execute pre-negotiated measurement logic as in active measurement apps. This challenge is further exacerbated due to the lack of TCB information for correlating packets for measurement.

C2: Unlike other apps that have a relatively "clean" measurement environment, the performance and accuracy of MopEye can be easily affected by measurement noises, because it has to relay packets for all applications in the phone.

To address challenge C1, we identify and correlate the correct packets for computing the RTT. Among the four types of TCP socket calls (i.e., `connect()`, `read()`, `write()`, and `close()`), our evaluation using `tcpdump` shows that the `connect()` call always accurately corresponds to a single round of packets, i.e., the SYN and SYN-ACK pair. That is, invoking a `connect()` call will immediately send out a SYN packet, and the call returns just after receiving a SYN-ACK packet. In contrast, a `read()`/`write()` call may involve multiple rounds of packet exchanges, and a `close()` call may not always elicit an ACK packet from the server.

However, it is difficult for MopEye to obtain the post-`connect()` timestamp accurately due to C2. Since MopEye uses non-blocking `SocketChannel` APIs to relay packets, it has to wait for the system's notification for a received ACK. This event-based notification can introduce an additional delay up to several milliseconds if there are other pending socket events (e.g., read/write or `VpnService`'s incoming packets). We resolve this inaccuracy problem by temporarily setting the socket into blocking mode for each `connect()` call. That is, MopEye runs a `connect()` call in a temporary new thread, which we call `socket-connect` thread. Once the connection is established, MopEye resumes the non-blocking mode and switches back to the main thread listening for read and write events. As a result, MopEye can obtain an accurate post-`connect()` timestamp for the RTT measurement and, at the same time, provides efficient packet relaying. As will be explained in §3, the temporary `socket-connect` threads also give us several other benefits for optimizing MopEye's performance.

Besides the TCP-based measurement, MopEye also supports DNS. Measuring the RTT for DNS is quite straightforward. We can obtain it by measuring the time between `send()` and `receive()` UDP socket calls, which correspond to DNS query and reply, respectively. However, obtaining an accurate post-`receive()` timestamp is still difficult because of C2. We adopt a similar solution by setting up a temporary thread for a blocking-mode measurement, except that this time we run the whole DNS processing, including DNS parsing and socket initialization, in the temporary thread (instead of just doing so for the `connect()` call as in the TCP measurement). This is because DNS is an application-layer protocol built upon UDP, and processing it should not block the main thread of `VpnService`.

3 Implementation and Enhancements

We have implemented MopEye in 11,786 LOC and deployed it to Google Play [6] on 16 May 2016 for a crowdsourcing measurement study[3]. Figure 4 presents the architecture of MopEye. It has three major components or core threads (created by our `MopEyeService` that extends the Android `VpnService` class). The `TunReader` and `TunWriter` threads handle read/write for the VPN tunnel, whereas the `MainWorker` thread is responsible for all the packet processing (i.e., packet parsing, mapping, and relaying) and RTT measurement.

In this section, we will detail how we solve the challenges of implementing `TunReader`, `TunWriter`, and `MainWorker`, particularly our methods of enhancing MopEye's performance. For better reading and quick reference, we include the subsection numbers in the corresponding components in Figure 4. Among them, §3.1 and §3.5.2 present solutions generic to all VPN-based apps on Android, whereas the rest can benefit various VPN-based traffic inspection systems on different OSes.

3.1 Zero-delay Packet Retrieval from the VPN Tunnel

Reading packets from the VPN tunnel is straightforward, but it is very challenging to fast-retrieve the packets under the existing Android VPN programming paradigm. To illustrate this problem, we use a code snippet from ToyVpn [19], a representative VPN client in the official Android SDK sample code. The code[4] shows a 100ms sleep before executing each `read()` call. The purpose of this sleep is to reduce CPU cycles for data reading. Therefore, the sleep period is determined by the tradeoff between CPU consumption and packet retrieval delay.

We are not aware of any solution addressing this delayed VPN read problem. The ToyVpn example [19] implements an "intelligent" sleeping algorithm to partially mitigate this problem. The basic idea is to stop sleeping when detecting consecutive packet reads. The recently proposed Haystack [42] adopts a similar idea, but the system performance is not acceptable, e.g., achieving

[3] IRB approval was obtained from Singapore Management University on 9 October 2015 under application IRB-15-093-A077(1015).

[4] Due to the page limit, we skip the code here and refer interested readers to `http://tinyurl.com/ToyVPN`.

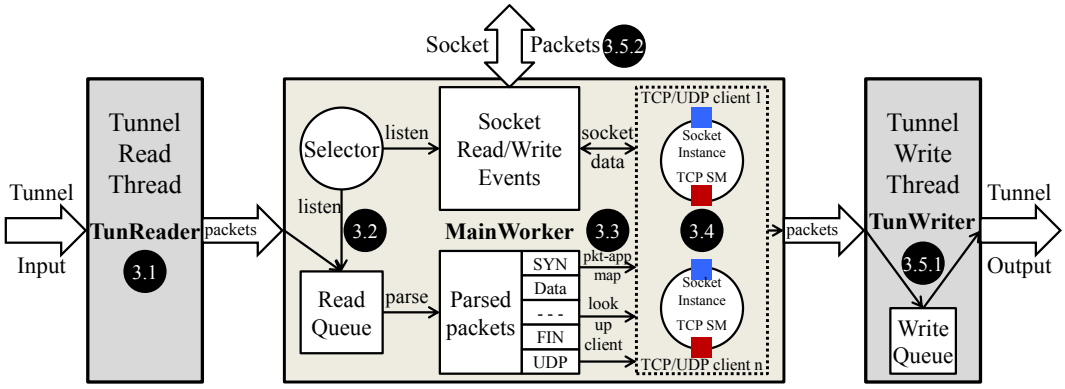


Figure 4: The architecture of MopEye.

only 17.2Mbps throughput from a 73Mbps upload link. PrivacyGuard [46], another system using `VpnService`, simply sets the sleep interval to 20ms.

We propose to fundamentally solve this problem by putting the VPN `read()` API into a blocking mode. That is, each `in.read()` call will be blocked until a packet is retrieved from the tunnel. This will effectively relieve the CPU from checking for data continuously. As a result, we must run the VPN `read()` API in a dedicated thread, i.e., `TunReader` in MopEye, and the retrieved packets will be put in a read queue shown in Figure 4.

Unfortunately, there is no API provided for setting the blocking mode of the VPN interface's file descriptor until Android 5.0. To implement our idea also for Android 4.0 to 4.4, we propose the following two solutions. First, at the native code level, we can invoke the `fcntl()` API with the `F_SETFL` command to set the blocking mode. Second, we can leverage Java reflection to invoke a non-API function called `setBlocking` in the unexported `libcore.io.IoUtils` class. We verify that this private function exists on Android from its inception.

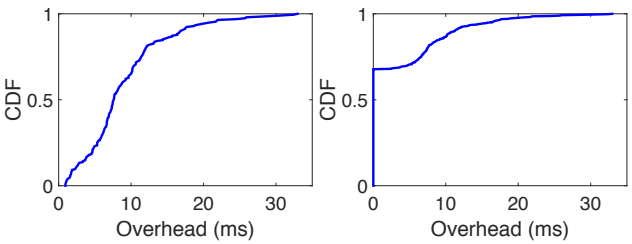
Although we can achieve zero-delay packet retrieval, there is a side effect of not being able to *timely* stop the `TunReader` thread in a blocking mode. We have tried the `Thread.interrupt()` API, but it does not work because in the absence of incoming packets the `read()` call will be blocked. To address this issue, we send a dummy packet to the VPN tunnel to release the blocked `read()` call. The dummy packet can be sent by MopEye itself for Android versions below 5.0. For Android 5.0+, however, MopEye no longer has the capability of letting its own packets go through the VPN tunnel due to the need of calling `addDisallowedApplication(mopeye)` to improve the performance (see §3.5.2). The only solution is to trigger a network request from other apps. After careful consideration, we use Android DownloadManager APIs [3] to stably trigger dummy download requests.

3.2 Monitoring Selector and Read Queue

As shown in Figure 4, we use a socket selector to listen for non-blocking read/write events from each socket instance and a read queue for receiving tunnel packets from `TunReader`. Being implemented as a single thread, `MainWorker`, however, cannot monitor both the socket selector and the tunnel read queue at the same time. To circumvent this problem, we leverage the existing `select()` waiting point to also monitor the read queue. That is, `TunReader` will issue a `Selector.wakeup()` event whenever it adds a new packet to the read queue. As a result, when the selector is woken up, MopEye will check for both socket and tunnel events, because either could have activated the selector. Moreover, to process the events timely, we interleave the code for checking these two types of events.

3.3 Lazy Packet-to-App Mapping

As presented in §2.2, MopEye performs a packet-to-app mapping for SYN packets in order to obtain per-app network performance. Our evaluation, however, shows that such mapping is expensive. Figure 5(a) shows the cumulative distribution function (CDF) of the overhead for parsing `/proc/net/tcp6|tcp` for each SYN packet. The experiment was performed on a Nexus 6 phone, containing 196 samples, and in the experiment we browsed a list of websites using the Chrome app. Over 75% of the samples required more than 5ms for the parsing; over 10% of them needed even more than 15ms. Furthermore, the overhead will increase with the number of active connections in the system.



(a) Before the lazy mapping. (b) After the lazy mapping.

Figure 5: CDF plots of packet-to-app overhead per packet.

We propose a *lazy* mapping mechanism to address this problem. First, we defer the mapping from the main thread to each temporary `socket-connect` thread mentioned in §2.4. Moreover, the mapping is performed

only after the connection is established or failed, thus not affecting the timely TCP handshake on the application side. Second and more importantly, we develop an efficient mapping algorithm that performs less `proc` file parsing. Our mapping algorithm is based on the observation that for multiple concurrent `socket-connect` threads, it is sufficient to let only one thread perform the parsing. Other threads just check and/or sleep to wait for the working thread to retrieve the mappings for them. We choose the sleep period of 50ms which is sufficiently large when compared with the parsing overhead in Figure 5(a). The evaluation results show that such a *lazy* mapping algorithm is very useful for scenarios like web browsing. For a total of 481 temporary `socket-connect` threads in a web browsing scenario, only 155 of them need to perform parsing. Moreover, the algorithm helps avoid the mapping overhead in the other 326 threads, i.e., achieving 67.8% mitigation rate as shown in Figure 5(b). Besides improving the mapping performance, it also helps reduce the CPU overhead.

Haystack [42] briefly mentions that they use *cache* to minimize the mapping overhead. However, cache-based mechanism could cause *inaccurate* packet-to-app mapping results. For example, both the Facebook app and accessing Facebook by Chrome may use the same server IP and port, but their mappings are different. This problem is more noticeable for advertisement modules since the same library may be embedded in many different mobile apps. Therefore, in order to obtain an accurate mapping, we use our own lazy mapping mechanism instead of the traditional cache-based mechanism.

3.4 Tuning TCP Performance

Besides implementing the basic user-space TCP/IP stack presented in §2.3, we have identified and tuned the following performance issues for fast packet relaying.

Maximum segment size (MSS) To maximize the throughput of the internal connections, MopEye sets the MSS option to 1460 bytes in the SYN/ACK packet and sends 1500-byte IP packets to the apps.

Receive window size Another factor affecting TCP throughput is the TCP receive window. MopEye assigns the maximum of 65,535 bytes to each MopEye's socket write and read buffer. We could also use the TCP window scale option [10] to further increase the throughput but have not done so, because the existing receive window is already big enough for achieving good performance and a bigger window size will increase the buffer memory.

No congestion and flow control Since no packet loss and reordering is expected in the VPN tunnel, MopEye forwards the data packets continuously to the app without waiting for the ACKs. Moreover, upon receiving a FIN/RST packet, MopEye stops the packet forwarding immediately.

Minimizing the use of expensive calls We try to minimize the use of expensive calls during the packet processing. For example, we discover that the `register()` call [1] for registering the socket selector can sometimes be very expensive. MopEye therefore executes this call in the `socket-connect` thread *only after* completing the handshaking for the internal connection. Other examples include never performing database operations in the main thread and always avoiding the debug log output.

3.5 Fast Dispatching of Tunnel and Socket Packets

3.5.1 Dispatching Packets to the VPN Tunnel

We observe that writing packets to the tunnel is *not* always fast, partially because multiple writing threads (e.g., `MainWorker` and individual `socket-connect` thread) share only one tunnel. We use the experimental results obtained from two writing schemes in Table 1 to illustrate this problem.

- *directWrite*: Writing is performed whenever there are packets to be sent to the tunnel.
- *queueWrite*: As illustrated in Figure 4, the packets are first put in a queue. A separate writing thread is used to output the packets. This scheme is currently adopted by MopEye.

	directWrite	queueWrite	oldPut	newPut
Total	1,244	2,161	810	5,321
0∼1ms	1,202	2,147	763	5,317
1∼2ms	30	12	39	1
2∼5ms	7	2	7	1
5∼10ms	3	0	1	2
>10ms	2	0	0	0

Table 1: Delay of writing packets to the VPN tunnel under four different writing schemes.

According to Table 1, the queueWrite scheme performs much better than the directWrite scheme. Among a total of 1,244 samples in the directWrite testing, we encounter 42 large writing overheads (i.e., those larger than 1ms). The corresponding result for the queueWrite testing is only 14 out of 2,161 samples. In particular, there are five extremely large overheads (i.e., those larger than 5ms) in the directWrite samples, two of which are even over 20ms. While there are still 14 overheads of 1∼5ms for queueWrite, they do not affect the performance of `MainWorker`, because they are performed by the dedicated `TunWriter` thread.

Although the queueWrite scheme significantly reduces the writing overhead, it introduces the overhead of packet enqueuing. We find that a traditional enqueuing scheme, denoted by oldPut, has large overheads. Among the 810 oldPut samples in Table 1, 47 have an overhead larger than 1ms. Our testing shows that most of the overheads between 1∼5ms are due to the

queue's `wait-notify` delay. When there are no packets in the queue, `TunWriter` goes to sleep by calling `queue.wait()` and is woken up by `queue.notify()`. We design a new enqueuing algorithm, denoted by newPut, to mitigate such delays. The basic idea is to let `TunWriter` perform more rounds of queue checking before going to `wait()`. Specifically, we design a sleep counter to systemize this process:

- The counter is initialized to 0 and is reset to 0 each time being woken up from `wait()`.
- When there are no packets in the queue, the counter increments for every round of checking and decrements (e.g., dividing by 2) whenever detecting a nonempty queue.
- `TunWriter` sleeps only when the counter reaches a threshold.

The newPut column in Table 1 shows the effectiveness of our algorithm. Out of the 5,321 samples, only four contain 1∼5ms overheads. Compared with the oldPut scheme, the percentage of large overheads drops from 5.69% to only 0.075%. It is worth noting that the remaining two large overheads of 5∼10ms are likely due to thread competition. We also observe that such competition effect is significantly reduced, because the enqueuing operation (at the microsecond level) is much faster than tunnel writing (at the 0.1ms level).

3.5.2 Dispatching of Socket Packets

When MopEye relays packets to the external connection, a delay overhead which could be up to several milliseconds comes from the `VpnService.protect(socket)` method [18]. Before establishing socket connections with remote app servers, MopEye must call the `protect(socket)` method to ensure that the socket packets will be sent directly to the underlying network. Without this method, the socket packets will be directed back to the VPN tunnel, thus creating a data loop.

Our solution is to replace the socket-wide `protect()` API with the application-wide `addDisallowedApplication()` API. By adding MopEye into the list of VPN-disallowed applications, we do not need to invoke `protect(socket)` for each socket client. Moreover, since MopEye just needs to call `addDisallowedApplication(mopeye)` once, the call is best invoked during the initialization of MopEye to avoid impact on `MainWorker`. The limitation of this solution is that `addDisallowedApplication()` is newly introduced in Android 5.0. For older versions, MopEye still has to call `protect(socket)`. Our mitigation method is to put `protect(socket)` in each `socket-connect` thread. In this way, only the performance of the SYN packet will be affected but not the subsequent data. Furthermore, this issue will be of less importance as more devices are upgraded to Android 5.0+, currently with over 60% of devices [2].

4 Evaluation

In this section, we present two sets of evaluation results. The first is on the measurement accuracy and overhead of MopEye, and the second is a set of crowdsourcing measurement results from 2,351 active users over nine months.

4.1 Measurement Accuracy and Overhead

4.1.1 Measurement Accuracy

The first evaluation we perform is on the accuracy of RTT measurement of MopEye. In addition to the standalone measurement, we also compare MopEye with MobiPerf v3.4.0 (the latest version at the time of our evaluation), which makes active network measurements using the state-of-the-art Mobilyzer library [40]. For a fair comparison, we use MobiPerf's HTTP ping measurement [37] because, like MopEye, it also uses SYN-ACK for the RTT measurement. For each destination, we use its raw IP address instead of the domain name so that MobiPerf's accuracy will not be interfered by DNS queries. Moreover, each result is presented by the mean of ten independent runs (MobiPerf does not provide detailed results of each run). We also run `tcpdump` to provide the reference measurement results.

Destinations	MopEye (mean, in ms)			MobiPerf (mean, in ms)		
	`tcp dump`	Mop Eye*	δ	`tcp dump`	Mobi Perf	δ
Google (`216.58.221.132`)	4.26	4	**0**	4.29	16.4	**12.11**
	4.47	5.5	**1.03**	4.35	18.5	**14.15**
	5.32	5	**0**	4.85	18	**13.15**
Facebook (`31.13.79.251`)	36.55	37	**0.45**	36.39	59.5	**23.11**
	36.55	37	**0.45**	36.72	55.2	**18.48**
	38.54	38.5	**0**	46.10	63.2	**17.10**
Dropbox (`108.160.166.126`)	284.85	284.5	**0**	361.76	409.7	**47.94**
	390.94	391	**0.06**	388.94	411.5	**22.56**
	513.78	513.5	**0**	395.87	475.2	**79.33**

* We round MopEye's μs-level results to *ms*-level, e.g., 4.135ms to 4ms.

Table 2: Measurement accuracy of MopEye and MobiPerf.

Table 2 presents three sets of results for Google, Facebook, and Dropbox, which experience RTTs on different scales. The differences between the RTT measurement of MopEye/MobiPerf and that of `tcpdump` are denoted by δ. The results clearly show that MopEye has a much better accuracy than MobiPerf—MopEye's measurement deviates from that of `tcpdump` by at most 1ms, whereas MobiPerf's deviations range from 12ms to 79ms. By assessing MobiPerf's code[5], we identify three factors responsible for MopEye's higher accuracy, including using the low-level socket call and the nanosecond-level timestamp method, and most importantly, putting the timing function just before and after the socket call. We refer interested readers to our previous poster version [48] for more details.

[5] `http://tinyurl.com/PingTask`, where HTTP ping starts from the line 438.

Throughput	Baseline	MopEye	Δ	Haystack	Δ
Download	24.47	24.01	**0.46**	20.19	**4.28**
Upload	25.97	25.08	**0.89**	6.79	**19.18**

Table 3: The download and upload throughput overhead of MopEye and Haystack.

4.1.2 Measurement Overhead

To measure the overhead introduced by MopEye, we first measure the additional delay introduced to the connection establishment and data transmission in other apps when MopEye is running. For the connection time, we implement a simple tool that invokes `connect()` to measure the time taken with and without MopEye. For data packets, we use the popular Ookla Speedtest app [16] to measure the latency. Both experiments are repeatedly executed on a Nexus 4 running Android 5.0. With a 95% confidence interval, the mean delay overhead of a round of SYN and SYN/ACK packets is 3.26∼4.27ms and that of data packets is 1.22∼2.18ms. Considering that the median of all 714,675 LTE RTTs in our dataset is 76ms, the delay overhead is acceptable.

Another important metric is the download and upload throughput overhead. We compare MopEye with Haystack [42] v1.0.0.8 (the latest version at the time of our evaluation), which uses the `VpnService` API to detect privacy leaks in app traffic. For a fair comparison, we do not enable Haystack's TLS traffic analysis for all its experiments. We use the Ookla Speedtest app as the reference tool to measure the throughput with and without MopEye/Haystack. All three experiments are repeatedly conducted in a dedicated WiFi network which provides very strong signal strength and stable throughput at around 25Mbps for both download and upload links.

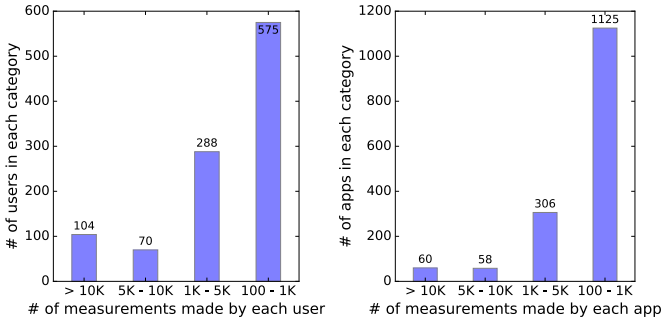
Table 3 presents the throughput results with Δ denoting the difference from our baseline using Speedtest. The results clearly show that MopEye achieves a much better throughput performance than Haystack. MopEye's throughput deviates from the baseline by less than 1Mbps, whereas that for Haystack ranges from 4Mbps (for the download link) to 19Mbps (for the upload link). In particular, we find that Haystack's throughput degrades significantly (e.g., 11.63Mbps for the download and 3.74Mbps for the upload) if we do not restart it for the next run. Therefore, in order to obtain the Haystack results in Table 3, we reset Haystack's VPN interface before each test. We attribute our superior results to the major challenges addressed in §3.

4.1.3 Resource Consumption Overhead

We now summarize the resource consumption overhead of MopEye and Haystack with a Nexus 6 playing a high-definition YouTube video for around one hour. According to Table 4, MopEye's resource consumption overhead is lower than that of Haystack in terms of CPU, battery, and memory. In particular, the CPU overhead with Haystack is over 9%, mainly because Haystack has to keep executing the VPN `read()` regardless there are app packets to be relayed or not. Moreover, we argue that the 1% battery overhead of MopEye is not contributed only by MopEye, because, with MopEye enabled, YouTube is no longer considered using the network interface by the system battery benchmark.

Scenario	Resource	Overhead	
		MopEye	Haystack
Playing a 58-minute high-definition (1080p) YouTube video	CPU	2.74%	9.56%
	Battery	1%	2%
	Memory	12MB	148MB

Table 4: The resource overhead of MopEye and Haystack.



(a) By user. (b) By app.

Figure 6: Number of measurements performed by each user/app that contribute to at least 100 measurements.

4.2 Crowdsourcing Measurement Results

Our MopEye deployment on Google Play has attracted 4,014 user installs from 126 countries since May 2016. In this section, we first describe the dataset used in this paper and then present our measurement analysis to underline the value of MopEye's opportunistic per-app measurement.

4.2.1 Dataset Statistics

By deploying MopEye for over ten months, to the best of our knowledge, we have collected the *first* large-scale per-app measurement dataset. Our analysis in this paper is based on the MopEye data received between its launch on 25 May 2016 and 3 January 2017. Our dataset covers a wide spectrum of devices, countries, and apps, and includes over 5 million RTT measurements.

User/Device coverage: The dataset includes a total of 2,351 devices that performed at least one measurement. Figure 6(a) shows the number of measurements performed by 1,037 devices each of which conducted at least 100 measurements. Although most of them are in the range of 100–1K, 462 of them (45%) contribute from 1K to more than 10K measurements each. This shows a significant number of consistently active users. Moreover, these user devices cover 922 different phone models, manufactueres of which include Samsung, HTC, LG, Motorola, Huawei, XiaoMi, and others. This evidences that MopEye can support a wide range of Android phones in the market.

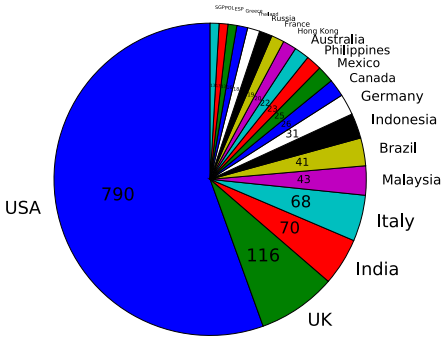


Figure 7: Distribution of the top 20 MopEye user countries.

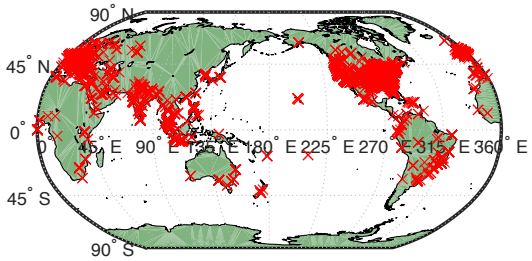
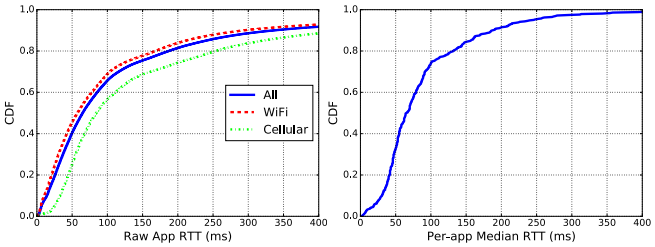


Figure 8: Locations of conducting the MopEye measurement.

Country distribution: Users in our dataset come from 114 countries worldwide. Figure 7 shows the distribution of the top 20 user countries, including the United States (790 users), United Kingdom (116 users), India (70 users), and Italy (68 users). Moreover, Figure 8 plots 6,987 geographical locations where the MopEye measurements were conducted. The figure visually shows that our dataset covers a large populated area, notably the North America, Europe, India, coastal regions of South America, Southeast Asia, and the Pacific Rim.

Applications measured: This dataset includes measurement on 6,266 apps. Figure 6(b) shows the distribution of the number of RTT measurements performed by each app that contributes at least 100 measurements, with a total of such 1,549 apps. Similar to Figure 6(a), most of them contribute 100–1K measurements, and 424 of them have between 1K and more than 10K measurements. The most popular (in terms of the number of times being measured) apps include social networking apps such as Facebook, Instagram, and WeChat, and system built-in apps such as YouTube and Google Play.

Measurements collected: The dataset contains a total number of 5,252,758 RTT measurements. Among them, 3,576,931 are measurements for TCP connections used by the apps, and the remaining 1,675,827 are for DNS measurements. Altogether they cover 106,182 destination IP addresses, 35,351 destination server domains, 2,427 destination server ports, and 943+ DNS servers. The most accessed domain is `graph.facebook.com` with 142,873 connections.



(a) All apps' raw RTTs. (b) Top 424 apps' median RTTs.

Figure 9: CDF plots of apps' raw RTTs and median RTTs.

4.2.2 Per-app Measurement Analysis

We now present the 3,576,931 per-app measurement results, which characterize the network performance experienced by different apps under different network types and ISPs in the wild. We envision ways of using the analysis results to improve the mobile network performance. For example, we reported our measurement results of WeChat to help Tencent (developer of the WeChat app) solve a misconfiguration problem [48].

Overall results. We first present the overall app performance in our dataset by plotting the distribution of apps' raw and median RTTs in Figure 9. Figure 9(a) shows the CDF plot of all 6,266 apps' raw RTTs, in which we further distinguish between WiFi and cellular access. Overall, the performance experienced by mobile users is good with a median RTT of 65ms (i.e., the value at the 0.5 line in Figure 9(a)). Moreover, ∼40% of the RTTs are below 50ms and ∼60% of the RTTs are below 100ms. However, we can still observe ∼20% of them suffering from relatively long RTTs (>200ms), and ∼10% at exceedingly long RTT (>400ms). In this dataset, WiFi shows superior performance than that on cellular networks. The median RTTs for WiFi, cellular networks (including 2G, 3G, and LTE), and LTE alone are 58ms, 84ms, and 76ms, respectively.

Figure 9(b) plots the median RTT distribution of 424 apps that have more than 1K measurements each (see Figure 6(b)). We choose the median over the mean value because the median is less affected by RTT outliers. The dataset also has enough measurements for each app, making the median a reliable measure. The figure shows that more than 70% of the apps experience less than 100ms in their RTTs. However, there are ∼10% of the apps suffering from more than 200ms of RTT.

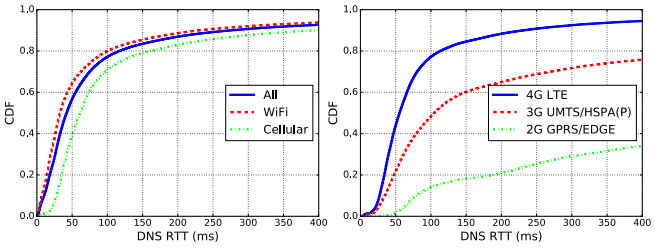
Representative apps' performance. We next study the network performance of representative apps that are frequently used in our daily life. Table 5 lists 16 such apps in five categories. For each app, we present its total number of RTT measurements and the median RTT. Most of these apps exhibit very good network performance. For example, Instagram, WeChat, Google Play Store, YouTube, and Amazon have a median RTT below 60ms. We also notice that the median RTT of Whatsapp is larger than 100ms. Next we present two case studies in more depth.

Category	Apps	# RTT	Median RTT
Social	Facebook	215,769	61ms
	Instagram	38,640	50.5ms
	Weibo	28,905	43ms
	Twitter	11,407	56ms
Commu-nication	WeChat	61,804	36ms
	Facebook Messenger	42,408	42ms
	Whatsapp	32,372	133ms
	Skype	16,264	76ms
Google	Google Play Store	100,115	48ms
	Google Play services	60,805	37ms
	Google Search	35,858	45ms
	Google Map	19,996	38ms
Video	YouTube	99,895	32ms
	Netflix	28,302	33ms
Shopping	Amazon	18,313	59ms
	Ebay	16,114	70ms

Table 5: Network performance of 16 representative apps.

Case 1: The vast majority of *.whatsapp.net domains do not perform well in many networks. Whatsapp employs a total of 334 `whatsapp.net` domains as its server domains, but the median RTT of all these domain traffic is as high as 261ms. Specifically, the median RTTs for all, except three, are larger than 200ms. The median RTTs for those three domains (starting with `mme`, `mmg`, or `pps`) are less than 100ms. According to our analysis, the three domains are deployed in the Facebook CDN, whereas the other 331 domains are with SoftLayer Technologies, a server hosting provider. Furthermore, we analyze the median RTTs on these 331 `whatsapp.net` domains in 20 most accessed networks (11 WiFi and 9 LTE networks) that have at least 100 measurements each. The results show that only two networks can achieve less than 100ms of RTT (77.5ms for a WiFi network and 56ms for the Verizon 4G network), six networks in the 100–200ms interval, eight networks in between 200ms and 300ms, and four networks with RTTs over 300ms. Moreover, our manual Ping tests from Singapore and Hong Kong to those domains report a latency of ∼250ms. All of the above show that there is much room for Whatsapp to improve their `whatsapp.net` network performance.

Case 2: Jio, India's largest 4G ISP, fails to provide acceptable performance to many app domains. In the course of analyzing the Whatsapp case, we find that Jio provides poor performance to many app server domains. Among all the ten 4G ISPs with more than 10K measurements, Jio is the only one that has a median RTT larger than 100ms. The median RTT of its 76,717 RTT measurements is as high as 281ms. Considering that the median RTT of its DNS measurements is only 59ms, the root cause lies very likely in its LTE core network. Moreover, our analysis of 115 domains (that have 100+ measurements each) in Jio finds that only 19 domains' median RTTs are less than 100ms, whereas the median RTTs of 67 domains are over 200ms, 57 domains over 300ms, and 24 domains even over 400ms. We further confirm that Jio's poor performance is not due to the performance of the app servers. It is because out of the 71 domains that have 100+ measurements each in both Jio and non-Jio LTE networks, 63 of them have much better latency (138ms less than Jio on average) with non-Jio LTE networks.



(a) All results. (b) Cellular results.

Figure 10: CDF plots of DNS measurement results.

4.2.3 DNS Measurement Analysis

Next we analyze the 1,675,827 DNS measurements received from 943+ WiFi and cellular DNS servers.

Overall results. Figure 10(a) shows the CDF plot of all measured DNS RTTs. According to the overall distribution, the DNS performance for mobile networks in the wild is good with a median of 42ms, and around 80% of DNS RTTs are less than 100ms. The DNS RTTs are in fact much better than the per-app performance by comparing Figure 10(a) with Figure 9(a). For example, 80% of per-app RTTs are less than 200ms, two times higher than DNS. This is probably because ISPs usually deploy local DNS servers. Additionally, we notice that WiFi's DNS RTTs are consistently lower than the overall results with a median of only 33ms; whereas that of cellular networks is 61ms. This indicates that the first-hop performance of WiFi is generally better than cellular networks.

We plot the detailed results for 2G, 3G, and 4G cellular networks in Figure 10(b). The CDF plots show clearly the performance difference among the three. More specifically, the median DNS RTT of 4G is 56ms; whereas that of 3G and 2G are as high as 105ms and 755ms, respectively. Most of the devices in our measurement use 4G—around 80% of DNS RTTs come from 4G. This also explains why the CDF plot for 4G DNS RTTs is close to that of all cellular RTTs.

Major 4G ISPs' DNS performance. We now take a closer look at the DNS performance of major 4G ISPs. Table 6 lists the performance of 15 LTE operators that have most DNS RTTs in our dataset. First, we notice that there is no clear correlation between the country and DNS performance. For example, the performance of most American ISPs, three Hong Kong ISPs, and two Malaysia ISPs are similar. Second, the majority of 4G ISPs provide good DNS performance with the median RTTs in 40-60ms. The only three outliers are the good-performer Singtel, and the poor-performers Cricket and U.S. Cellular. To gain a better understanding, we further study these three ISPs along with Verizon, a representative of other ISPs.

ISP Name	Country	# RTT	Median RTT
Verizon	America	80,227	46ms
Jio 4G	India	52,397	59ms
AT&T	America	51,421	53ms
Singtel	Singapore	34,609	27ms
Boost Mobile	America	21,854	50ms
Sprint	America	20,878	51ms
3	HK (China)	14,354	53ms
MetroPCS	America	13,282	60ms
T-Mobile	America	9,084	45ms
CMHK	HK (China)	5,820	50ms
Celcom	Malaysia	4,120	56ms
CSL	HK (China)	3,099	61ms
Cricket	America	2,822	93ms
Maxis	Malaysia	2,419	40ms
U.S. Cellular	America	1,988	76ms

Table 6: DNS performance of 15 LTE 4G operators.

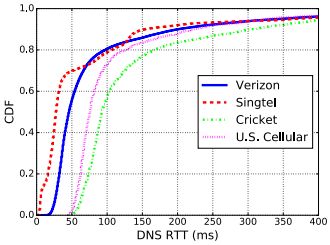


Figure 11: CDF plots for DNS performance of four LTE ISPs.

Figure 11 presents the DNS RTT distribution of the four selected ISPs with the Verizon plot as the baseline. The plots show that Singtel has an outstanding first-hop performance with 5,084 DNS RTTs less than 10ms (14.7% of its total RTTs), whereas Verizon has less than 1% of its DNS RTTs below 10ms. This is mainly because Singtel has deployed the latest upgrade of LTE, Tri-band 4G+ [21], countrywide [12]. On the other hand, the DNS performance of Cricket and U.S. Cellular clearly is worse than the baseline. In particular, the minimum RTTs of Cricket and U.S. Cellular are around 43ms, much higher than the best performance of Singtel and Verizon. They are probably using the pre-4G or near-4G implementations, because we find that around half of their DNS RTTs (64% of Cricket and 45% of U.S. Cellular) are still from non-LTE networks.

Key Takeaway: MopEye enables a large-scale deployment of per-app measurements in the wild, which help understand and diagnose the network quality of app providers and mobile networks at different granularity.

5 Related Work

Many measurement tools have been proposed to understand mobile network performance. They could be classified into crowdsourcing measurement apps (e.g., [30, 28, 25, 40]) and controlled testbeds (e.g., [47, 35, 24]). They study 3G/4G networks' RRC (Radio Resource Control) state dynamics [41, 28, 44], analyze the behaviors of cellular networks [27, 49, 33, 29], measure mobile network performance and reliability [40, 35, 22, 23, 45], and perform other measurements [37, 38]. MopEye belongs to the domain of crowdsourcing measurements. Using the `VpnService` API to perform passive network measurement, MopEye is the first app that provides per-app network performance on unrooted phones without user intervention. With MopEye, we also provide the first report of large-scale per-app network measurements.

Recently, researchers are interested in utilizing the `VpnService` API for different purposes. Nearly all of them focus on detecting privacy leakage [26] by relaying and intercepting mobile apps' traffic either in the smartphone [46, 42] or at a remote VPN server [36, 43]. Two recent works [34, 39] use a remote VPN server to identify traffic differentiation and optimize traffic volume in cellular networks. MopEye is different from all these related works in that we leverage the `VpnService` API for per-app network performance measurement. Indeed, MopEye is the first and the only one on the market that provides per-app measurement for end users. Moreover, our solutions for tackling the delayed VPN read problem (§3.1) and mitigating the VPN `protect()` delay (§3.5.2) can benefit all VPN-based apps, such as OpenVPN [9].

Due to the traffic-interception capability of `VpnService` APIs, it is important for VPN-based apps to preserve users' privacy in their design. Unfortunately, many VPN apps on the market fail to do so according to a recent study [31]. The majority of them use remote VPN servers for traffic relay, but not always in a secure fashion (e.g., no encryption for the tunnel to VPN servers, or no tunneling for DNS traffic). In contrast, our MopEye adopts the local phone-side traffic forwarding scheme, without additional risks associated with VPN servers, such as leaking user traffic. Further, unlike PrivacyGuard [46] and Haystack that perform traffic *content* inspection, MopEye makes no such attempt, let alone the TLS interception performed by those two. This may be an important factor contributing to a much higher number of MopEye installs than Haystack, which reached only 1.5K installs by the end of March in 2017 [4].

6 Conclusion

In this paper we proposed MopEye, a novel measurement app to monitor per-app network performance on unrooted smartphones. By leveraging the `VpnService` API on Android to intercept all network traffic, MopEye was able to opportunistically measure each app for its network RTT without network overhead and user intervention. We overcame a number of challenges to achieve a fast packet relaying and an accurate measurement in MopEye. We have deployed MopEye to Google Play for an IRB-approved crowdsourcing study for over ten months. By collecting and analyzing the first large-scale per-app measurement dataset, we discovered a number of new findings on the per-app and DNS network performance experienced by real users in the wild. We plan to further improve MopEye (e.g., supporting more metrics beyond RTT), and release more analysis results for app developers and ISPs to optimize their performance.

References

[1] AbstractSelectableChannel selector() — Android Developers. http://developer.android.com/reference/java/nio/channels/spi/AbstractSelectableChannel.html#register(java.nio.channels.Selector,int,java.lang.Object).

[2] Dashboards — Android Developers. https://developer.android.com/about/dashboards/index.html.

[3] DownloadManager — Android Developers. https://developer.android.com/reference/android/app/DownloadManager.html.

[4] Lumen Privacy Monitor has reached 1.5K installs on Google Play! https://twitter.com/lumen_app/status/845230899226689537.

[5] MobiPerf on Google Play. https://play.google.com/store/apps/details?id=com.mobiperf.

[6] MopEye on Google Play. https://play.google.com/store/apps/details?id=com.mopeye.

[7] Netalyzr on Google Play. https://play.google.com/store/apps/details?id=edu.berkeley.icsi.netalyzr.android.

[8] Network Speed Test on Windows Store. http://www.windowsphone.com/en-us/store/app/network-speed-test/9b9ae06b-2961-41ef-987d-b09567cffe70.

[9] OpenVPN - Open Source VPN. https://openvpn.net/.

[10] RFC 1323 - TCP Extensions for High Performance. http://tools.ietf.org/html/rfc1323.

[11] RFC 793 - Transmission Control Protocol. https://tools.ietf.org/html/rfc793.

[12] Singtel's 4G Network Deployment History. https://www.singtel.com/personal/i/4g/why-singtel.

[13] SpeedChecker on Google Play. https://play.google.com/store/apps/details?id=uk.co.broadbandspeedchecker.

[14] Speedtest X HD WiFi & Mobile Speed Test on App Store. https://itunes.apple.com/us/app/speedtest-x-hd-wifi-mobile/id366593092.

[15] Speedtest.net on App Store. https://itunes.apple.com/us/app/speedtest.net-mobile-speed/id300704847.

[16] Speedtest.net on Google Play. https://play.google.com/store/apps/details?id=org.zwanoo.android.speedtest.

[17] Speedtest.net on Windows Store. http://www.windowsphone.com/en-us/store/app/speedtest-net/4fcd4de1-050b-44dc-b123-a786808eb49b.

[18] The protect() API in VpnService — Android Developers. http://developer.android.com/reference/android/net/VpnService.html#protect(int).

[19] ToyVpn. https://github.com/android/platform_development/tree/master/samples/ToyVpn.

[20] VpnService — Android Developers. http://developer.android.com/reference/android/net/VpnService.html.

[21] What are 4G, 4G+ and Tri-band 4G+? https://www.singtel.com/personal/i/4g/support.

[22] BALTRUNAS, D., ELMOKASHFI, A., AND KVALBEIN, A. Measuring the reliability of mobile broadband networks. In *Proc. ACM IMC* (2014).

[23] BALTRUNAS, D., ELMOKASHFI, A., AND KVALBEIN, A. Dissecting packet loss in mobile broadband networks from the edge. In *Proc. IEEE INFOCOM* (2015).

[24] CHEN, Q., LUO, H., ROSEN, S., MAO, Z., IYER, K., HUI, J., SONTINENI, K., AND LAU, K. QoE Doctor: Diagnosing mobile app QoE with automated UI control and cross-layer analysis. In *Proc. ACM IMC* (2014).

[25] DENG, S., NETRAVALI, R., SIVARAMAN, A., AND BALAKRISHNAN, H. WiFi, LTE, or both?: Measuring multi-homed wireless Internet performance. In *Proc. ACM IMC* (2014).

[26] ENCK, W., GILBERT, P., CHUN, B., COX, L., JUNG, J., MCDANIEL, P., AND SHETH, A. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. Usenix OSDI* (2010).

[27] FALAKI, H., LYMBEROPOULOS, D., MAHAJAN, R., KANDULA, S., AND ESTRIN, D. A first look at traffic on smartphones. In *Proc. ACM IMC* (2010).

[28] HUANG, J., QIAN, F., GERBER, A., MAO, Z., SEN, S., AND SPATSCHECK, O. A close examination of performance and power characteristics of 4G LTE networks. In *Proc. ACM MobiSys* (2012).

[29] HUANG, J., QIAN, F., GUO, Y., ZHOU, Y., XU, Q., MAO, Z., SEN, S., AND SPATSCHECK, O. An in-depth study of LTE: Effect of network protocol and application behavior on performance. In *Proc. ACM SIGCOMM* (2013).

[30] HUANG, J., XU, Q., TIWANA, B., MAO, Z., ZHANG, M., AND BAHL, P. Anatomizing application performance differences on smartphones. In *Proc. ACM MobiSys* (2010).

[31] IKRAM, M., VALLINA-RODRIGUEZ, N., SENEVIRATNE, S., KAAFAR, M. A., AND PAXSON, V. An analysis of the privacy and security risks of Android VPN permission-enabled apps. In *Proc. ACM IMC* (2016).

[32] JENKOV, J. Java NIO Selector. http://tutorials.jenkov.com/java-nio/selectors.html.

[33] JIANG, H., LIU, Z., WANG, Y., LEE, K., AND RHEE, I. Understanding bufferbloat in cellular networks. In *Proc. ACM CellNet* (2011).

[34] KAKHKI, A., RAZAGHPANAH, A., LI, A., KOO, H., GOLANI, R., CHOFFNES, D., GILL, P., AND MISLOVE, A. Identifying traffic differentiation in mobile networks. In *Proc. ACM IMC* (2015).

[35] KVALBEIN, A., BALTRUNAS, D., EVENSEN, K., XIANG, J., ELMOKASHFI, A., AND OLIVEIRA, S. The Nornet Edge platform for mobile broadband measurements. In *Computer Networks* (2014).

[36] LE, A., VARMARKEN, J., LANGHOFF, S., SHUBA, A., GJOKA, M., AND MARKOPOULOU, A. AntMonitor: A system for monitoring from mobile devices. In *ACM SIGCOMM Workshop on Crowdsourcing and Crowdsharing of Big Internet Data (C2BID)* (2015).

[37] LI, W., MOK, R., WU, D., AND CHANG, R. On the accuracy of smartphone-based mobile network measurement. In *Proc. IEEE INFOCOM* (2015).

[38] LI, W., WU, D., CHANG, R., AND MOK, R. K. P. Demystifying and puncturing the inflated delay in smartphone-based WiFi network measurement. In *Proc. ACM CoNEXT* (2016).

[39] LI, Z., WANG, W., XU, T., ZHONG, X., LI, X.-Y., LIU, Y., WILSON, C., AND ZHAO, B. Y. Exploring cross-application cellular traffic optimization with Baidu TrafficGuard. In *Proc. USENIX NSDI* (2016).

[40] NIKRAVESH, A., YAO, H., XU, S., CHOFFNES, D., AND MAO, Z. Mobilyzer: An open platform for controllable mobile network measurements. In *Proc. ACM MobiSys* (2015).

[41] QIAN, F., WANG, Z., GERBER, A., MAO, Z., SEN, S., AND SPATSCHECK, O. Characterizing radio resource allocation for 3G networks. In *Proc. ACM IMC* (2010).

[42] RAZAGHPANAH, A., VALLINA-RODRIGUEZ, N., SUNDARESAN, S., KREIBICH, C., GILL, P., ALLMAN, M., AND PAXSON, V. Haystack: In situ mobile traffic analysis in user space. *CoRR abs/1510.01419* (2015).

[43] REN, J., RAO, A., LINDORFER, M., LEGOUT, A., AND CHOFFNES, D. R. ReCon: Revealing and controlling PII leaks in mobile network traffic. In *Proc. ACM MobiSys* (2016).

[44] ROSEN, S., LUO, H., CHEN, Q., MAO, Z., HUI, J., DRAKE, A., AND LAU, K. Discovering fine-grained RRC state dynamics and performance impacts in cellular networks. In *Proc. ACM MobiCom* (2014).

[45] RULA, J. P., AND BUSTAMANTE, F. E. Behind the curtain: Cellular DNS and content replica selection. In *Proc. ACM IMC* (2014).

[46] SONG, Y., AND HENGARTNER, U. PrivacyGuard: A VPN-based platform to detect information leakage on Android devices. In *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)* (2015).

[47] WEI, X., GOMEZ, L., NEAMTIU, I., AND FALOUTSOS, M. ProfileDroid: multi-layer profiling of Android applications. In *Proc. ACM MobiCom* (2012).

[48] WU, D., LI, W., CHANG, R., AND GAO, D. MopEye: Monitoring per-app network performance with zero measurement traffic. In *Proc. ACM CoNEXT Student Workshop* (2015).

[49] XU, Q., ERMAN, J., GERBER, A., MAO, Z., PANG, J., AND VENKATARAMAN, S. Identifying diverse usage behaviors of smartphone apps. In *Proc. ACM IMC* (2011).

Emu: Rapid Prototyping of Networking Services

Nik Sultana[†], Salvator Galea[†], David Greaves[†], Marcin Wójcik[†], Jonny Shipton[†], Richard G. Clegg[‡], Luo Mai[§], Pietro Bressana[*], Robert Soulé[*], Richard Mortier[†], Paolo Costa[♯], Peter Pietzuch[§], Jon Crowcroft[†], Andrew W. Moore[†], Noa Zilberman[†]

[†]*University of Cambridge,* [‡]*Queen Mary University of London,*
[§]*Imperial College London,* [*]*University of Lugano,* [♯]*Microsoft Research*

Abstract

Due to their performance and flexibility, FPGAs are an attractive platform for the execution of network functions. It has been a challenge for a long time though to make FPGA programming accessible to a large audience of developers. An appealing solution is to compile code from a general-purpose language to hardware using high-level synthesis. Unfortunately, current approaches to implement rich network functionality are insufficient because they lack: (i) libraries with abstractions for common network operations and data structures, (ii) bindings to the underlying "substrate" on the FPGA, and (iii) debugging and profiling support.

This paper describes *Emu*, a new standard library for an FPGA hardware compiler that enables developers to rapidly create and deploy network functionality. Emu allows for high-performance designs without being bound to particular packet processing paradigms. Furthermore, it supports running the same programs on CPUs, in Mininet, and on FPGAs, providing a better development environment that includes advanced debugging capabilities. We demonstrate that network functions implemented using Emu have only negligible resource and performance overheads compared with natively-written hardware versions.

1 Introduction

FPGAs are an attractive platform for implementing network functions. They combine the flexibility of software with the performance and predictability of hardware. Major cloud service providers, such as Microsoft, Baidu, and Amazon, already deploy FPGAs in their data centers to accelerate internal and third-party workloads [36, 40], and implement custom network services [8, 34].

Consequently, there has been significant interest in developing tools and techniques to simplify FPGA programming and making FPGAs accessible to a larger fraction of developers. A common approach is to use *high-level synthesis* (HLS), which allows developers to program FPGAs using a general-purpose language (GPL) such as C, which is then compiled to a hardware description language (HDL), such as Verilog or VHDL.

Unfortunately, while high-level synthesis undoubtedly simplifies FPGA development, HLS alone is not sufficient to implement rich network functionality. Notably, developers who wish to target FPGAs lack three key components. First, they need library support comparable to that in normal software programming, i.e., they need access to re-usable modules and libraries that provide abstractions for common functions and data structures. Second, they require a binding to the underlying "substrate" on the hardware. Unlike CPUs, on an FPGA, there are usually no operating system (OS) and drivers mediating access to hardware. Finally, they need support for fine-grained debugging capabilities, akin to what is available to software developers today.

We present *Emu*, a framework for network functions on FPGAs. Emu builds on the Kiwi compiler [43] that allows computational scientists to program FPGAs with .NET code. The relationship with Emu to .NET/Kiwi is roughly analogous to that of the stdlib to C/GCC—Emu provides the implementation for essential network functionality. Emu and HLS thus result in a powerful substrate for developers to rapidly implement and deploy network functions using a high-level language.

Moreover, Emu virtualizes the hardware context of the network pipeline, allowing developers to write code that is portable across different heterogeneous targets. Our current implementation supports CPUs, simulation environments, and FPGAs. Using Emu, developers can run their network functions as normal processes, using virtual or real NICs, and using network simulators, simpli-

fying debugging and testing. Emu also offers debugging and profiling tools that enable developers to inspect the behavior of the application at runtime.

While simplifying development is important, most network operators are not willing to sacrifice performance for ease-of-development. With Emu, developers can have both: Emu supports designs with different performance metrics such as bandwidth, latency, or operations-per-second.

Using Emu, we have created various prototype implementations of networking services, ranging from an L2 switch to a high-performance Memcached server [17]. Each service is expressed in C#, which can be transformed to host or FPGA instantiations. The FPGA-centered code, created from the C# compiler output and transformed into Verilog executes, for our prototype, on a NetFPGA SUME card [49].

Domain-specific languages such as P4 [5] or ClickNP [26] are too low-level and are designed to support only specific tasks, e.g., packet processing. In contrast, Emu enables the development of a broader set of services, leveraging its support for general-purpose programming.

We compare the performance of Emu against software-only and native Verilog implementations (§5). Our results show that Emu-generated code significantly outperforms software-only versions in terms of latency, latency variance, and throughput, while having a negligible overhead compared to native implementations.

Overall, this paper makes the following contributions:

1. a *"standard library"* for network services, which allows hardware network functions that go beyond header processing to be written in C#. This enables dynamic, conditional processing for network services such as DNS and Memcached. The framework can be customized for different performance metrics, and we illustrate the tradeoffs involved;
2. an *execution environment* that supports running a single codebase over heterogeneous targets, including CPUs, network simulators, and FPGAs; and
3. *debugging support* that translates high-level idioms for debugging, profiling, and monitoring into a low-level language for controlling runtime program state.

Emu and all datasets used in this paper are publicly available [15], and our FPGA designs will be contributed to the NetFPGA community.

2 Motivation

The goal of Emu is to make it easy for software developers with no expertise in hardware languages to quickly develop, test, and deploy network services on an FPGA. Using Emu, application developers can offload network logic to hardware with only modest effort.

The main reason for moving network services from the CPU to FPGAs is increased performance, as demonstrated by existing applications [24, 46, 47]. Moving network functions to hardware also saves CPU cycles, which would otherwise be spent in polling the network interface card (NIC), as typically done in high-performance packet-processing frameworks such as DPDK [52] or netmap [37].

Different data center services, however, have different performance goals. Some applications are throughput-sensitive, e.g., a streaming service, while for others latency is the primary concern [11]. Further, in some cases, latency can be an indirect contributor to low application performance [21]. For example, in Memcached, even tens of microseconds are sufficient to drop the number of queries-per-second significantly [50]. By providing a set of suitable abstractions and APIs, Emu allows developers to optimize towards their preferred performance metric such as ease-of-coding, throughput, or latency.

Our approach can be seen as an example of network paravirtualization: it allows high-level network primitives to be compiled to the paravirtualized hardware (e.g., FPGA or CPU) via the Emu framework. This has the potential to foster innovation at the NIC level, with vendors adding custom logic to natively support some of our high-level APIs. Our library can then be extended to map API calls such as those communicating packets, or doing novel data manipulation (e.g., match-action table processing such as longest-prefix matching, hash and checksum computation, and other conditional operations at line-rate) to custom hardware blocks when available and to rely on paravirtualization, otherwise.

While many consider the translation from a general-purpose language to a hardware language to be the main challenge, there is another important obstacle, namely providing support for debugging an application. Debugging FPGA programs requires the use of hardware-level simulators [32, 42] or probing tools [12], and most network service developers are unfamiliar with these tools. Emu addresses this problem on two levels: (i) it allows application code to be run in a x86 runtime environment. This enables developers to verify and debug the functionality of their code, speeding up the development process; (ii) it provides debugging, monitoring, and profiling tools for the application while running on the hardware. It does this by offering familiar GPL-like abstractions, fitting application developers' capabilities.

Previous work tried to address only a subset of these challenges, as we summarize in Table 1. Past solu-

Solution	What is it?	Target Applications	Processing Paradigm	Language	Performance Metric	Debug Environment[1]	Compiler to Verilog
Emu	"Standard library"	Networking applications	Any	.NET	User defined (see §3.2)	x86, Mininet and Emu env.	Kiwi
Kiwi	Compiler and libraries	Scientific applications	Any	.NET	Execution time/area	x86	Kiwi
Vivado HLS	Compiler and libraries	Scientific applications	Any	C, C++, System C	Throughput	C simulation	Vivado HLS
SDNet	Programming environment	Networking applications	Packet processing	PX/P4	Throughput	C++ simulation	SDNet
P4	Programming language	Networking applications	Packet processing	P4	Throughput	P4 behavioral simulator, Mininet	P4 compiler, then P4FPGA/SDNet.
ClickNP	Programming language/model	Networking applications	Packet processing	ClickNP	Throughput	Undefined	ClickNP, then Altera OpenCL or Vivado HLS

[1]Excluding RTL simulators, accessible on the HDL level to all solutions

Table 1: Comparison between different representative solutions for enabling networking services in hardware

tions either focus on packet processing applications exclusively (e.g., P4 [5] and ClickNP [26]) or target scientific applications only without providing abstractions appropriate for networking (e.g., Kiwi and Vivado HLS). Emu tackles both challenges, while combining additional advantages: supporting a heterogeneous debug environment, as well as user-defined optimizations for performance.

Interestingly, while Vivado HLS specifies latency as a performance metric, this refers to increasing parallelism within the design, rather than network latency metrics. An example showing how increasing parallelism adds to latency is given in §5.3.

3 Emu framework

With Emu, developers can use a general-purpose language to implement high-performance network functions that run on FPGAs. The Emu runtime provides an abstract target environment, and a library of functionality that can execute on both CPUs and FPGAs, simplifying debugging and deployment. Moreover, Emu provides an interface to intellectual property (IP) blocks, i.e., specialized modules that take advantage of hardware features (§3.4). This further abstracts away the details of hardware development. Next, we present an overview of the Emu framework, and describe a typical workflow.

3.1 Background

Emu combines and extends several existing components, including the Kiwi compiler for HLS, and NetFPGA [49] as a hardware target. Note that Emu is not strictly dependent on these specific components—one could use a different HLS compiler or network-attached FPGA.

Kiwi. Originally designed to support scientific computing applications, the Kiwi compiler transforms the target language of .NET compilers, i.e., the common intermediate language (CIL), into a register-transfer level (RTL) description of hardware in Verilog [43]. The Verilog output can then be used to configure FPGAs. We apply Kiwi to the domain of network services and extend it to support networking operations. Emu provides a library to facilitate the development of network functions, and includes some improvements to Kiwi as described in §3.2.

NetFPGA SUME [49] is the latest generation in the NetFPGA family, and provides a low-cost, FPGA-centered PCIe hardware platform for research and education. Alongside several packet-centered reference projects (e.g., an IPv4 router, Ethernet switch, and NIC), NetFPGA has provided the base platform to prototype a variety of high-performance hardware designs for network-centered applications, the best example being prototype hardware for OpenFlow SDN [33].

3.2 Kiwi extensions

Emu provides the following functionality on top of Kiwi: (i) we add support for IP blocks, as defined in §3.4. Although Emu readily generates instances of various components, such as RAMs, ALUs and format converters, we add new support for easily instantiating other IP blocks; (ii) the second extension is needed to mix hard

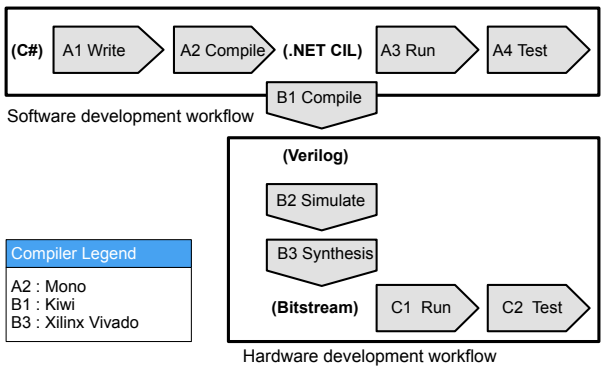


Figure 1: Components of the Emu framework

and soft timing. Kiwi is designed for scientific acceleration, giving it complete freedom over the schedule of operations, which is especially important for multi-cycle floating-point ALU operations. To support the hard timing, cycle-accurate, requirements of network services, Kiwi's scheduler is adapted and paused in parts of the design; (iii) a third extension is the support for casting a byte or a word array into a struct, so that various bit fields take on names and types. C# supports this in the `unsafe` dialect, but the KiwiC version used by Emu only accepts the strongly-typed safe subset of C#; (iv) finally, the largest primitive datatype in C# is the 64-bit word. To achieve higher performance, we require wider I/O busses. Emu defines user types for larger words and provides overloads for all of the arithmetic operators needed.

3.3 Emu overview

Figure 1 shows the main components of the Emu framework, which include: (i) a library tailored to network functions; (ii) runtime support for running C#-coded network programs on a CPU; and (iii) library support for developing and debugging programs. Steps A1, A2, A3 and A4 show the standard C# compilation to bytecode and running/testing on a CPU. B1 uses Kiwi (and Emu extensions) to compile from .NET CIL to Verilog. Steps B2 and B3 (using the NetFPGA framework and the Xilinx compiler) output a bitstream that can be executed on NetFPGA, and this is run and tested in hardware in steps C1 and C2.

Emu extends Kiwi by offering library support customized to the networking domain. Kiwi also provides a "substrate" to support programs that it compiles—the substrate serves as a runtime library for those programs and Emu extends this substrate.

Developers describe a network service in terms of what it does to packets sent and received through net-

```
1  // If the frame does not contain an IPv4 packet then we do
       not set its output port; this implicitly drops the
       frame.
2  if (dataplane.tdata.EtherType_Is(EtherTypes.IPv4))
3  {
4    // Configure the metadata such that if we have a hit
         then set the appropriate output port in the
         metadata, otherwise broadcast.
5    if (dstmac_lut_hit) {
6      NetFPGA.Set_Output_Port(ref dataplane, lut_element_op)
           ;
7    } else {
8      NetFPGA.Broadcast(ref dataplane);
9    }
10 }
11 Kiwi.Pause();
12
13 // Add source MAC to our LUT if it's not already there,
       thus the switch "learns".
14 if (!srcmac_lut_exist)
15 {
16   LUT[free] = srcmac_port;
17   free = (free > (LUT_SIZE - 1)) ? 0 : free++;
18 }
```

Figure 2: Part of a switch implementation, showing use of our API for protocols (Line 2) and NetFPGA (Line 6)

work logical ports, which are attached at runtime to network interfaces made available by the OS. The interfaces may be physical or virtual (e.g., a tap device). Emu provides a library and runtime support so developers can quickly test prototypes of network functions written in high-level languages. Layers of abstraction between the .NET runtime and the OS provide virtual/physical network interfaces. By using virtual interfaces, developers can test network functions in a simulator.

3.4 Library features

Basic usage. Emu extends the C# code that can be compiled by Kiwi with a library of functions that provide convenience (e.g., by defining frequently-used protocol formats) and performance (e.g, by providing access to carefully crafted IP blocks, see below). Thus any C# code that can be compiled by Kiwi can be used in Emu. An example snippet from our implementation of a switch is provided in Figure 2. Most of the code is standard C#, except for line 11, which controls Kiwi's scheduling (see below), and lines 2 and 6, which use utility functions.

Protocol parsing. Parsers for commonly-used packet formats are available for reuse. As an example, Figure 3 shows the code to instantiate some of the parsers used in the NAT implementation (§4.4). All parsers that may be needed during runtime are instantiated on loading, and, as the snippet shows, it can handle TCP over IPv4 over Ethernet, as well as ARP over Ethernet.

Writing new parsers for custom protocols is straight-

```
var eth = new EthernetWrapper(dataplane.tdata);
var ip = new IPv4Wrapper(dataplane.tdata);
var tcp = new TCPWrapper(dataplane.tdata);
var arp = new ARPWrapper(dataplane.tdata);
```

Figure 3: Parsers for different protocol formats

```
public uint DestinationIPAddress
  { get { return BitUtil.Get32( ips, 0); }
    set { BitUtil.Set32(ref ips, 0, value); } }

public uint SourceIPAddress
  { get { return BitUtil.Get32( ips, 4); }
    set { BitUtil.Set32(ref ips, 4, value); } }
```

Figure 4: Parsing IPv4 headers

forward. Figure 4 shows how two IPv4 fields are manipulated using standard C# programming style as well the utility functions `BitUtil.Get32` and `BitUtil.Set32`.

Using IP blocks. While C# provides an easy development environment, to maximize the performance of a design, it is sometime recommended to use specialized IP blocks that take advantage of the hardware capabilities, such content addressable memory (CAM) used in some of our implementations. These blocks are accessible through the facilities of Kiwi, as mentioned in §3.2.

An example use of an IP block is a hashing module. Figure 5 shows the C# implementation of the protocol required to seed a value (when the hash is used in streaming mode). The protocol involves two signals, `init_hash_ready` and `init_hash_enable`, used for handshaking, and a bundle of eight signals `data_in` used for sending a byte to the core. We can implement the handling of arbitrary protocols in C#, and this enables us to interface with any IP block.

Multi-threading and scheduling control. Kiwi reinterprets concurrency primitives that are used when programming software to improve its hardware generation. It provides a thread-based concurrency library with two type of semantics: (i) software semantics reduces to concurrency primitives provided by .NET, while (ii) hardware semantics forms logical circuits in which parallel threads may be wired into parallel logical sub-circuits.

Using these types of semantics, .NET programs may be executed on general-purpose x86 CPUs by using the software semantics, or on FPGAs by using the logical circuit semantics. In the latter case, Kiwi produces descriptions with much finer parallelism than what is possible on software platforms, whose parallelism is at most instruction-level. We take advantage of this and further refine it to achieve maximal pipelining of projects.

For high performance, developers can also aid Kiwi in scheduling computations across time using annotations,

```
public static void Seed(byte data_in)
 {
   while (init_hash_ready) { Kiwi.Pause(); }
   PearsonHash.data_in = data_in;
   init_hash_enable = true;
   Kiwi.Pause();
   while (!init_hash_ready) { Kiwi.Pause(); }
   Kiwi.Pause();
   init_hash_enable = false;
   Kiwi.Pause();
 }
```

Figure 5: Part of the wrapper for our hashing module

```
// Extract the frame from NetFPGA_Data into a byte array.
public static void Get_Frame (NetFPGA_Data src, ref byte[]
      dst)
...

// Move the contents of a byte array into the frame field
      in NetFPGA_Data.
public static void Set_Frame (byte[] src, ref NetFPGA_Data
      dst)
...

// Read the input port (i.e., port on which we received the
       frame).
public static uint Read_Input_Port (NetFPGA_Data dataplane)
...

// Set the output port to a specific value. (i.e., the port
       to which we are forwarding the frame.)
public static void Set_Output_Port (ref NetFPGA_Data
      dataplane, ulong value)
...
```

Figure 6: Utility functions for interacting with the FPGA dataplane

as shown in line 11 in Figure 2. This breaks up computation and allows Kiwi to schedule a suitable amount of computation in a single clock cycle by providing a cycle-accurate notion where needed. If Kiwi schedules too little computation, it is inefficient; if it schedules too much computation, the implementation on the target FPGA device fails. Currently, Kiwi is target oblivious, i.e., it does not have information about clock rates.

Utility functions. In addition to the purpose-specific APIs described in previous sections, Emu also includes general utility functions. These form a library of C# code and are intended to help abstract unnecessary details, such as the functions listed in Figure 6 for interacting with the FPGA target. One could have different sets of such functions for different targets, e.g., without changing the code for protocol parsing or IP blocks.

3.5 Debug-related features

Emu produces a debug environment by the systematic extension of programs to interpret *direction commands* at runtime to enable debugging, monitoring and profil-

```
1 if V_trace_idx < max_trace_idx then
2   V_trace_buf[V_trace_idx] := V;
3   inc V_trace_idx;
4   continue
5 else
6   inc V_trace_overflow;
7   break
```

Figure 7: Code that implements the direction command "trace *X* max_trace_idx" (If the buffer is not full, the new value of X is logged, the index incremented, and control is returned to the program that hosts this code; otherwise, it indicates depletion of the associated buffer resource and break the program's execution.)

ing. This design came about after we found ourselves extending our ad hoc debugging and monitoring code to support additional features. It is inspired by the commands found in profilers and debuggers such as gdb.

Emu uses a language of direction commands [44]. Figure 7 describes one such command, "trace *V* max_trace_idx", and shows how to express this high-level direction command as a program executable by a controller, with which Emu programs are extended (see Figure 8).

Table 2 lists other supported high-level direction commands. Commands are translated into programs that execute on a simple controller embedded in the program. We model the controller as a *counters, arrays, and stored procedures* (CASP) machine, which refers to the constituents of the machine's memory.

Extending a C# program to support direction commands involves inserting (i) named *extension points* with runtime-modifiable code in a computationally weak language (no recursion); and (ii) *state* used for bookkeeping by that code to implement direction features.

Debugging can also be conducted using *direction packets*. Direction packets are network packets in a custom and simple packet format, whose payload consists of (i) code to be executed by the controller; or (ii) status replies from the controller to the director. It enables us to remotely direct a running program, similar to gdb's "remote serial protocol" [18].

Emu minimizes the overhead that these features introduce at runtime by extending a program (before compilation) to support the precise set of required debugging or profiling features. This frugality does not come at the cost of inflexibility, however, because the extension points at runtime can be reconfigured to perform different debugging or profiling functions.

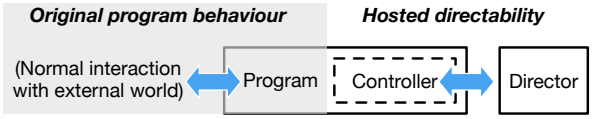


Figure 8: *Controller* embedded into the program, acting as the agent of the *director* (The director and controller exchange commands and their outputs.)

3.6 Limitations

The main limitation of Emu when compared to HDLs is the lack of low-level control over hardware designs, and here Emu is partly limited by Kiwi's capabilities. Kiwi does not yet allow one to internalize instances of an HDL module, and this forces Emu to interface with such modules instead of instantiating them.

In addition, Emu currently supports only a limited number of protocols, but developers can extend the library to support more protocols (see Figure 4).

Finally, depending on the required performance, developers must be aware of the hardware that the design is deployed on, or is interfacing with. For example, for a given throughput, a wider I/O bus may be required.

4 Use cases

We have implemented different networking services to demonstrate the benefits of Emu. These include forwarding (§4.1), measurement and monitoring (§4.2), performance sensitive applications (§4.3), and more complex applications such as NAT and caching (§4.4). The use cases cover a range of network services, and can include bespoke features, e.g., encryption schemes. The use case implementations are available under an open-source license [15]. Some of the applications are also available as contributed projects to the NetFPGA community, starting with NetFPGA SUME release 1.4.0.

4.1 Packet forwarding

Learning switch. We implement a standard layer-2 learning switch, similar in functionality to the NetFPGA SUME reference switch [45]. Beyond header processing, which is a basic networking function, it provides an example of how content addressable memory (CAM) is implemented in Emu, and how a native FPGA IP CAM block can be used. While the first option does not burden developers with implementation details, the latter provides better resource usage and timing performance. A simplified version of our implementation is shown in Figure 2. The full version is around 150 lines of C#, and the resulting Verilog is around 500 lines.

Command	Behaviour
print X	Print the value of variable X from the source program.
break L $\langle B \rangle$	Activate a (conditional) breakpoint at the position of label L.
unbreak L	Deactivate a breakpoint.
backtrace $\langle \$ \rangle$	Print the "function call stack".
watch X $\langle B \rangle$	Break when X is updated and satisfies a given condition.
unwatch X	Cancel the effect of the "watch" command.
count { reads X $\langle B \rangle$ $\langle \$ \rangle$ / writes X $\langle B \rangle$ $\langle \$ \rangle$ / calls *fname* $\langle B \rangle$ $\langle \$ \rangle$ }	Count the reads or writes to a variable X, or the calls to a function *fname*.
trace { start X $\langle B \rangle$ $\langle \$ \rangle$	Trace a variable, subject to a satisfied condition, and up to some length.
trace { stop X	Stop tracing a variable.
trace { clear X	Clear a variable's trace buffer.
trace { print X	Print the contents of a variable's trace buffer.
trace { full X	Check if a variable's trace buffer is full.

Table 2: Directing commands (Note that count has similar subcommands to those of trace.)

L3–L4 filter. We provide a tool that emulates the command-line parameter interface of IP tables [35]. Instead of modifying a Linux server's filters, it generates code that slots into our learning switch. This turns the switch into a L3 filter over sets of IP addresses or protocols (ICMP, UDP, and TCP), or an L4 filter over ranges of TCP or UDP ports.

4.2 Measurement and monitoring

ICMP echo. We have implemented an ICMP echo server to obtain two baselines: (i) a *qualitative* baseline on the difficulty of implementing a simple network server, and (ii) a *quantitative* baseline on how much time is saved by avoiding the system bus, CPU, OS, and network stack.

TCP ping. Sometimes the network handles ICMP traffic differently to the protocols used by applications such as TCP and HTTP. For example, a faulty configuration of the network may discard packets on some TCP ports on a machine, but without affecting the reachability of that machine through ICMP [22]. TCP ping involves a simple reachability test by using the first two steps of the three-way connection setup handshake. It is thus a more complex extension of ICMP echo. Our implementation is around 700 lines of C#, and the resulting Verilog is around 1,200 lines.

4.3 Performance-sensitive applications

DNS. We provide a simple DNS server that supports non-recursive queries. Our prototype supports resolution queries from names (of length at most 26 bytes) to IPv4 addresses, but these constraints can be relaxed to handle longer names and IPv6. If the queried name is absent from the resolution table, the server informs the client that it cannot resolve the name. Our implementation is around 700 lines of C#, and the resulting Verilog around 1,200 lines. [1]

Memcached [17] is a well-known distributed in-memory key/value store that caches read results in memory to quickly respond to queries. Its protocol uses a number of basic commands such as `GET` (to retrieve a value associated with the provided key), `SET` (to store a key/value pair) and `DELETE` (to remove a key/value pair), and supports both ASCII and binary protocols.

Memcached is sensitive to latency, and even an extra 20 μs are enough to lose 25% throughput [50]. Our initial Memcached implementation with Emu focussed on latency only and therefore supported only a limited version of the protocol, allowing only `GET`/`SET`/`DELETE` using the binary protocol over UDP, with 6-byte keys and 8-byte values. We later experimented with different extensions of this design, adding support for the ASCII protocol, larger key/value sizes, and for the use of DRAM and multiple CPU cores. These features introduce different trade-offs with respect to latency, throughput, and functionality.

4.4 Other applications

NAT. We provide a network address translation (NAT) service, supporting both UDP and TCP, which was im-

[1] It is a coincidence that the code length is the same as for the TCP ping use case.

```
public class Data {
  public bool matched = false;
  public ulong result = 0;
}
public class LRU {
  public static Data Lookup(ulong key_in) {
    Data res = new Data();
    ulong idx = HashCAM.Read(key_in);
    if (HashCAM.matched) {
      res.matched = HashCAM.matched;
      res.result = NaughtyQ.Read(idx);
      NaughtyQ.BackOfQ(idx);
    }
    return res;
  }
  public static void Cache(ulong key_in, ulong value_in) {
    ulong idx = NaughtyQ.Enlist(value_in);
    HashCAM.Write(key_in, idx);
  }
}
```

Figure 9: Least-recently-used (LRU) cache in Emu

plemented by a second-year undergraduate student. The implementation is written entirely in C#, without the use of Verilog-based cores, and has less than 1,000 lines.

One of the advantages of Emu is that the same code can run on multiple platforms, enabling a better development cycle. We use the NAT service as a test case, compiling it to three different targets: software, Mininet [31], and hardware.

Caching. One potential application that can benefit from offloading to hardware is caching. For example, SwitchKV [27] uses SDN-enabled switches to dynamically route read requests to a cache if content is available. This idea can be extended to directly implement a cache in the data plane, reducing load on storage servers. Implementing a cache in a DSL such as P4, however, would be difficult, because the eviction logic must be managed by the control plane. In contrast, with Emu, one can easily implement a look-aside, least-recently-used (LRU) cache in a few lines, as shown in Figure 9.

5 Evaluation

Our evaluation of Emu has the following aims: (a) provide evidence that using Emu is beneficial in terms of resources and performance, compared with other solutions; and (b) explore if Emu can be used to implement high-performance network services.

5.1 FPGA hardware

At the core of the NetFPGA SUME board is a Xilinx Virtex-7 690T FPGA device. The memory subsystem combines both static random access memory (SRAM) and dynamic random access memory (DRAM). It supports up to 32 GB of RAM, and can run as a stand

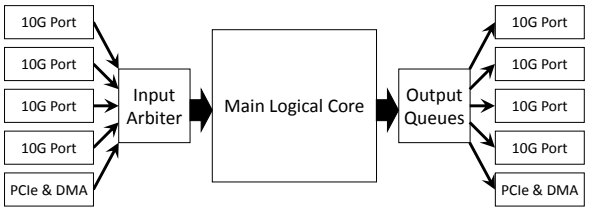


Figure 10: NetFPGA reference pipeline (The input arbiter, logical core, and output queues form the data plane.)

alone computing unit [23]. NetFPGA SUME's native frequency is 200 MHz.

The NetFPGA reference designs share a generic FPGA architecture, shown in Figure 10, with multiple physical interfaces surrounding a logical data-path. Emu capitalizes on this generic NetFPGA design: we target only the main logical core and build upon all other components to be shared between services, thus requiring no hardware expertise.

5.2 Experimental setup

Our experiments are conducted using a server with a single 3.5 GHz Intel Xeon E5-2637 v4 CPU with 64 GB DDR4 memory and a SuperMicro X10-DRG-Q motherboard. The machine runs Ubuntu Linux 14.04 LTS with the kernel version 3.13.0-106-generic. It has a dual port 10 GbE NIC (Intel 82599ES). The machine also includes a NetFPGA SUME board for the performance comparison. We use an Endace DAG 9.2X2 card for accurate latency measurements. All traffic is captured by the DAG card and used to measure the latency of the device-under-test (DUT) alone. The latency of the setup itself is measured first and deducted from all subsequent measurements. For latency measurements, the server runs the service pinned to a single CPU core with a warm cache.

For our throughput measurements, we use the Open Source Network Tester (OSNT) [1] as the traffic source. OSNT replays real traffic traces while modifying traffic rate to find the maximum throughput (e.g. queries per second). When testing, the server is configured to achieve maximum throughput (e.g. using multiple CPU cores), and this configuration changes between tests.

5.3 Comparison against hardware services

Next, we evaluate the immediate overheads of using Emu and show that the resulting implementations are comparable with native HDL designs.

	Emu	NetFPGA reference	P4FPGA
Logic resources	3509	2836	24161
Memory resources	118	87	236
Module latency	8 cycles	6 cycles	85 cycles
Throughput (Mpps)	59.52	59.52	53

Table 3: Comparison between Emu switch (C#), NetFPGA reference switch (Verilog), and P4FPGA switch (P4), using 64 byte packets

We compare the Emu learning switch, written in C# and compiled using Kiwi, with the NetFPGA SUME reference switch written directly in Verilog. We further extend this comparison to a similar design, written in P4 and compiled to NetFPGA SUME [47]. We do not compare with SDNet [39], as done by Dang et al. [10], because P4FPGA has better reported performance. As previous work [47], we use 256-entry tables.

Table 3 shows the resources consumed by the main logical core in each design. These results confirm that the resource overhead is minimal, making Emu an attractive solution. Furthermore, out of the reported resources consumed by Emu core, 85% are used by the CAM, which is an IP block, and only 15% by the C# generated logic. We note that, in all our use cases, the FPGA resources are never exhausted, and consume less than 33% of the logic resources, including the debug controller.

In terms of latency, Emu has only a minor overhead over the main logical core in the NetFPGA SUME reference switch design. In comparison to P4FPGA, Emu provides much lower latency than the compared design, mostly because Emu is not bounded by the match/action paradigm. In terms of throughput, instead, while P4FPGA achieves 53 Mpps for 64 byte packets using a 250 MHz clock, and a header parser for every port, Emu achieves full line rate (59.52 Mpps) using a 200 MHz clock and a single header parser.

Unfortunately, the authors of ClickNP [26] do not provide enough information, such as the FPGA clock rate, which would allow for a fair comparison with Emu. However, their reported packet-processing rate for similar applications (e.g., a firewall with 56 Mpps) is on par with Emu, as is the latency (e.g., 11 cycles for L4 Parser). In terms of resource usage, ClickNP has a resource utilization of 0.9× compared with the NetFPGA reference design's header parser (resp. 3.2× for a multi-threaded design). Emu's resource utilization, instead, is 0.7× with a single-thread design (1.2× with a multi-thread design).

5.4 Comparison against software services

In the previous section, we compared against equivalent implementations running on FPGAs. Now, we explore the performance of the different use cases from §4 against software-based, Linux native counterparts.

Setup. ICMP Echo and TCP Ping are used to evaluate the performance of a simple networking operation. We measure the round-trip time (RTT) required to reply to a request of the DUT alone. Latency measurements are performed for 100K packets. We configure NAT as a gateway to/from the local network, and measure the latency between an input interface from the external network and an output interface to the local one.

The Memcached evaluation uses the memaslap benchmark [30], configured to use a mix of 90% `GET` and 10% `SET` requests with random keys. The Emu Memcached implementation uses UDP and the ASCII protocol. We compare against a Linux Memcached server with 4 threads and 64 MB of memory, also running the UDP and ASCII protocols.

Results. We show the latency and throughput results in Table 4. Across all use cases, Emu achieves a reduction in latency from one to three orders of magnitude. Most importantly, unlike the host-based implementations, Emu's services exhibit a very short tail latency. This is particularly important as in distributed applications the application performance is often bound by the tail latency [11]. This means that not only Emu yields very low latency but it also guarantees *predictable* performance. In contrast, host-based implementations suffer from unpredictable delays and interrupts across the stack and exhibit a much higher variability with the tail-to-average ratio variying from 1.09 to 2.98 (resp. from 1.02 to 1.04 for Emu).

Emu also significantly outperforms host-based solution in term of throughput with improvements ranging from a factor of 2.1 up to a factor of 5.2. Interestingly, these results were obtained using a single-threaded Emu's configuration and could be further improved by instantiating multiple Emu cores. For example, in the Memcached usecase, using four Emu cores (one per port) further increases by 3.7× when considering a workload of 90% `GET` and 10% `SET` requests. SET requests must be applied to all instances, thus their relative ratio in performance cannot improve. The downside is that such an approach requires changes to the main logical core wrapper in NetFPGA SUME.

Optimizations. Further extending the above use cases can be done in different ways. For Memcached, it is possible to increase the memory available to Emu, using ei-

Network service	Emu Average latency (μs)	Emu 99th-perc. latency (μs)	Emu Throughput (million queries/s)	Host Average latency (μs)	Host 99th-perc. latency (μs)	Host Throughput (million queries/s)
ICMP Echo	1.09	1.11	3.226	12.28	22.63	1.068
TCP Ping	1.27	1.29	2.105	21.79	65.00	1.012
DNS	1.82	1.86	1.176	126.46	138.33	0.226
NAT	1.32	1.34	2.439	2444.76	6185.27	1.037
Memcached	1.21	1.26	1.932	24.29	28.65	0.876

Table 4: Comparison between services running on a host and Emu-based services (C#)

ther on-board or on-chip memory. On-board memory, e.g., using the DDR3 DRAM memory modules on NetFPGA SUME, has a size advantage, but the disadvantage of increased and variable latency (e.g., due to DRAM refreshes); on-chip memory has the benefit of low, constant latency, but is of smaller size. While NetFPGA SUME has 51 MB of on-chip memory, devices such as Xilinx Ultrascale+ have up to 65 Gbit on-chip, providing a solution at much larger scale. Further scaling can be achieved by using the Emu-based design as a (large) L1 cache, bounded to a few GBs, where cache misses are sent to a host [46] and implemented using the NetFPGA.

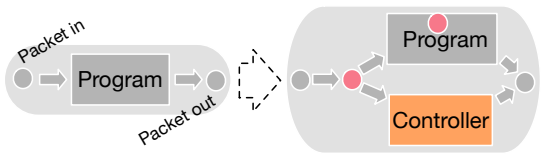


Figure 11: Transformation of the program to include a controller (Normal packets are handled without change, but *direction packet* are passed to the controller. Pink dots represent *extension points*, one of which is added within the control flow of the original program.)

5.5 Debugging

We extend the DNS and Memcached use cases in two ways: (i) adding code to check if a received packet is a *direction packet* intended for the controller (see Figure 11), in which case the controller (and not the original program) processes the packet; (ii) adding an extension point in the body of the (DNS or Memcached) main loop, allowing us to influence and observe the program from that point onwards. We form an enumerated type that corresponds to the program variables whose values the controller may access and change. The code for each value of the enumerated type refers to the program value, e.g., instructing the controller to increment it.

We evaluate Emu's debug environment by carrying out a quantitative analysis of the impact that the controller has on the program in which it is embedded. We measure this impact in terms of *utilization* of resources on the FPGA and the *performance* of the host program.

Table 5 shows the utilization and performance for DNS and Memcached, respectively, extended with different controller features: reading, writing, and incrementing a variable. The impact on utilization and performance is small, and dominated by the controller logic, rather than specific-purpose and runtime-programmable registers. Utilization improvements are due to the optimization process during the place-and-route state in hardware generation; occasionally this results in more utilization-efficient allocations.

An example of using directed packets is the debug process of our Memcached implementation. The Memcached service running on hardware replied with an error message, while no problem was detected in simulation. Using directed packets, we examined the Memcached service: directing the packets to report the checksum calculated within Emu revealed a bug in the checksum implementation and simulation environment.

5.6 Summary

Our evaluation demonstrates the advantages of Emu: (i) hardware resource usage is significantly lower than that of other approaches, adding only modest overhead when compared with bespoke HDL-only designs; (ii) the latency overhead is small compared to HDL designs and is similar to or better than that of other baselines; (iii) the overhead from the debug extensions is negligible, making Emu an attractive debug environment.

Our results also show an important advantage of Emu over host-based solutions: while absolute performance always depends on the CPU cores, memory bandwidth and frequency, FPGAs enjoy the benefit of predictability.

Artefact	Utilization (%)	Performance (%)	
	Logic	Latency	Queries-per-sec
DNS	100.0	100.0	100.0
+**R**	103.4	100.0	100.0
+**W**	115.1	99.5	100.0
+**I**	109.8	99.5	100.0
Memcached	100.0	100.0	100.0
+**R**	99.2	100.0	100.0
+**W**	99.8	100.5	100.0
+**I**	100.6	100.0	100.0

Table 5: Profile of utilization and performance (**R**ead, **W**rite, and **I**ncrement are instructions supported by the controller. Latency is compared at the 99th percentile.)

The median latency of our designs is both 10× lower than the median of the host-based solutions, with a small variance. While the difference between the median and 99th percentile is less than 200 ns for Emu, for host-based designs the variance is in the order of microseconds to tens of microseconds. This not only improves RTT and flow-completion times, but it also enables users to better schedule resources as they know when a reply is due.

6 Related work

FPGAs are increasingly deployed inside data centers, and their performance is getting closer to specialized hardware [51]. Recently there has been a large body of work on how to offload critical network and application services to FPGAs [2, 13, 14, 16, 24, 25, 36, 41, 48]. All of these proposals, however, leverage HDLs, making them unsuitable for the majority of developers who lack hardware skills. Emu addresses this issue by removing most of the challenges related to hardware programming and making FPGAs accessible to non-hardware experts, while retaining high performance.

We are not the first to target this goal and in the past there have been many efforts to make programming FPGAs easier, e.g., using a DSL [6, 7, 9, 38, 39], including network-specific ones [3, 5, 26]. These DSLs typically have a narrow scope and limit the performance or ability to implement certain network services. For example, P4 [5] is a popular DSL for packet processing that supports compilation to hardware including FPGAs. However, it is only applicable to tasks that can be processed by parse-match-action style systems. LINQits [9] proposes a tool chain that compiles an embedded query language (LINQ) into various platforms, including FPGAs, but it is specialized for answering queries and would require considerable adaptation to perform networking tasks. In contrast, Emu does not restrict the set of network services that can be implemented and offers a more general programming environment.

High-level synthesis (HLS) tools [28] generate HDL from high-level languages such as Scala, or Java (using Lime [4]), but they do not offer specific support for network programming. One exception is the Maxeler MPC-N system [29], which provides a "dataflow engine" to offload network computations to hardware. The engine runs kernels that are programmed using a subset of Java, and proprietary tooling. This approach, however, targets a proprietary hardware platform and lacks the ability to run seamlessly on both CPU and FPGAs. Conversely, Emu makes few assumptions about the underlying hardware and can be ported to different FPGAs. In addition, Emu's support for executing programs on a CPU and in simulation, combined with its advanced monitoring and profiling capabilities, greatly simplifies debugging of network programs.

The work in this paper is based on Kiwi [20, 43]. In previous work, Kiwi was used to distribute an application across network-connected hosts [19], but the network-related code was simple and had to be written from the ground up, because it lacked the "standard library" abstractions and debugging support provided by Emu.

7 Conclusion

Although the performance and availability of programmable network hardware has increased, making effective use of it remains beyond the reach of most developers. We have presented Emu, a framework that enables application developers to write network services in a high-level language (C#) and have them automatically compiled to execute across a number of platforms, including traditional CPUs (x86), simulation environments (Mininet), and an FPGA platform (NetFPGA), without compromising on performance.

We showed that the performance of Emu-based network services exceeds software-based solutions and is on par with native HDL implementations. Implementations on Emu permits services to run on different targets, support better debug capabilities and allow for easier transition of workloads among targets.

Acknowledgements. We thank Gordon Brebner, Han Wang, Matthew P. Grosvenor, the anonymous reviewers, and our shepherd, Christopher Rossbach. We acknowledge the support from the UK Engineering and Physical Sciences Research Council (EPSRC) (EP/K032968/1, EP/K034723/1, EP/K031724/2, and an UROP grant), Leverhulme Trust (ECF-2016-289) and Newton Trust, EU H2020 SSICLOPS (644866), SNF (166132) and a Google Faculty Research Award.

References

[1] Gianni Antichi, Muhammad Shahbaz, Yilong Geng, Noa Zilberman, Adam Covington, Marc Bruyere, Nick McKeown, Nick Feamster, Bob Felderman, Michaela Blott, Andrew W. Moore, and Philippe Owezarski. OSNT: Open source network tester. *IEEE Network*, 28(5):6–12, 2014.

[2] Shadi Atalla, Andrea Bianco, Robert Birke, and Lucado Giraudo. NetFPGA-based load balancer for a multi-stage router architecture. In *World Congress on Computer Applications and Information Systems*, pages 1–6. IEEE, Jan 2014.

[3] Michael Attig and Gordon Brebner. 400 Gb/s Programmable Packet Parsing on a Single FPGA. In *Symposium on Architectures for Networking and Communications Systems*, pages 12–23. IEEE Computer Society, 2011.

[4] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures. In *Object Oriented Programming Systems Languages and Applications*, pages 89–108. ACM, 2010.

[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[6] Gordon Brebner and Weirong Jiang. High-speed packet processing using reconfigurable computing. *IEEE Micro*, 34(1):8–18, 2014.

[7] Kevin J Brown, Arvind K Sujeeth, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Parallel Architectures and Compilation Techniques*, pages 89–100. IEEE, Oct 2011.

[8] Adrian Caulfield, Eric Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A Cloud-Scale Acceleration Architecture. In *International Symposium on Microarchitecture*. IEEE, Oct 2016.

[9] Eric S. Chung, John D. Davis, and Jaewon Lee. Linqits: Big data on little clients. *SIGARCH Comput. Archit. News*, 41(3):261–272, June 2013.

[10] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki-Suh Lee, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. Network hardware-accelerated consensus. *CoRR*, abs/1605.05619, 2016. URL: `http://arxiv.org/abs/1605.05619`.

[11] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

[12] Vivado Hardware Debug. `https://www.xilinx.com/products/design-tools/vivado/debug.html`.

[13] Ken Eguro. Automated Dynamic Reconfiguration for High-Performance Regular Expression Searching. In *International Conference on Field-Programmable Technology*. IEEE, Dec 2009.

[14] Ken Eguro and Ramarathnam Venkatesan. FPGAs for trusted cloud computing. In *International Conference on Field-Programmable Logic and Applications*. IEEE, Aug 2012.

[15] Emu Project. `http://www.cl.cam.ac.uk/research/srg/netos/projects/emu/`.

[16] Felix Engelmann, Thomas Lukaseder, Benjamin Erb, Rens van der Heijden, and Frank Kargl. Dynamic packet-filtering in high-speed networks using NetFPGAs. In *Int. Conf. on Future Generation Communication Technologies*, pages 55–59. IEEE, Aug 2014.

[17] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124), 2004.

[18] GDB Remote Serial Protocol. `http://www.embecosm.com/appnotes/ean4/embecosm-howto-rsp-server-ean4-issue-2.html`.

[19] David Greaves and Satnam Singh. Distributing C# methods and threads over Ethernet-connected FPGAs using Kiwi. In *International Conference on Formal Methods and Models for Codesign*, pages 1–9. IEEE, July 2011.

[20] David J. Greaves and Satnam Singh. Designing application specific circuits with concurrent C# programs. In *Formal Methods and Models for Codesign*, pages 21–30. IEEE, July 2010.

[21] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can jump them! In *Symposium on Networked Systems Design and Implementation*, pages 1–14. USENIX Association, 2015.

[22] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. *SIGCOMM Computer Communication Review*, 45(4):139–152, August 2015.

[23] Jong Hun Han, Noa Zilberman, Bjoern A. Zeeb, Andreas Fiessler, and Andrew W. Moore. Prototyping RISC based, reconfigurable networking applications in open source. *CoRR*, abs/1612.05547, 2016. URL: `http://arxiv.org/abs/1612.05547`.

[24] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a Box: Inexpensive Coordination in Hardware. In *Symposium on Networked Systems Design and Implementation*, pages 425–438. USENIX Association, 2016.

[25] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High Performance Packet Processing with FlexNIC. In *Architectural Support for Programming Languages and Operating Systems*, pages 67–81. ACM, 2016.

[26] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *SIGCOMM*, pages 1–14. ACM, 2016.

[27] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be fast, cheap and in control with switchkv. In *Symposium on Networked Systems Design and Implementation*, pages 31–44. USENIX Association, March 2016.

[28] Grant Martin and Gary Smith. High-Level Synthesis: Past, Present, and Future. *IEEE Design Test of Computers*, 26(4):18–25, July 2009.

[29] Maxeler MPC-N Series. `https://www.maxeler.com/products/mpc-nseries/`.

[30] memaslap - Load testing and benchmarking a server. `http://docs.libmemcached.org/bin/memaslap.html`.

[31] Mininet. `http://mininet.org/`.

[32] ModelSim. `https://www.mentor.com/products/fv/modelsim/`.

[33] Jad Naous, David Erickson, Adam Covington, Guido Appenzeller, and Nick McKeown. Implementing an OpenFlow switch on the NetFPGA platform. In *Symposium on Networked Systems Design and Implementation*, pages 1–9. ACM, 2008.

[34] Jian Ouyang, Shiding Lin, Wei Qi, Yong Wang, Bo Yu, and Song Jiang. SDA: Software-defined accelerator for large-scale DNN systems. In *Hot Chips Symposium*, pages 1–23. IEEE, Aug 2014.

[35] Gregor N Purdy. *Linux iptables Pocket Reference*. O'Reilly Media, 2004.

[36] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Int. Symp. on Computer Architecture*, pages 13–24. IEEE, Jun 2014.

[37] Luigi Rizzo. Netmap: A Novel Framework for Fast Packet I/O. In *USENIX Annual Technical Conference*. USENIX Association, 2012.

[38] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: a Compiler and Runtime for Heterogeneous Systems. In *Symposium on Operating Systems Principles*. ACM, 2013.

[39] SDNet. `https://www.xilinx.com/products/design-tools/software-zone/sdnet.html`.

[40] Amazon Web Services. EC2 Instances (F1) with Programmable Hardware. `https://goo.gl/fmEQPK`.

[41] David Sidler, Gustavo Alonso, Michaela Blott, Kimon Karras, Kees Vissers, and Raymond Carley. Scalable 10Gbps TCP/IP stack architecture for reconfigurable hardware. In *Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 36–43. IEEE, 2015.

[42] Vivado Simulator. `https://www.xilinx.com/products/design-tools/vivado/simulator.html`.

[43] Satnam Singh and David J. Greaves. Kiwi: Synthesis of FPGA Circuits from Parallel Programs. In *Field-Programmable Custom Computing Machines*, pages 3–12. IEEE, April 2008.

[44] Nik Sultana, Salvator Galea, David Greaves, Marcin Wójcik, Noa Zilberman, Richard Clegg, Luo Mai, Richard Mortier, Peter Pietzuch, Jon Crowcroft, and Andrew W. Moore. Extending programs with debug-related features, with application to hardware development. *CoRR*, abs/1705.09902, 2017. URL: `http://arxiv.org/abs/1705.09902`.

[45] NetFPGA SUME Reference Switch. `https://github.com/NetFPGA/NetFPGA-SUME-public/wiki/NetFPGA-SUME-Reference-Learning-Switch`.

[46] Yuta Tokusashi and Hiroki Matsutani. A Multilevel NOSQL Cache Design Combining In-NIC and In-Kernel Caches. In *Symposium on High-Performance Interconnects*, pages 60–67. IEEE, Aug 2016.

[47] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. P4FPGA : A Rapid Prototyping Framework for P4. In *Proceedings of the Symposium on SDN Research*, pages 122–135. ACM, April 2017.

[48] Louis Woods, Jens Teubner, and Gustavo Alonso. Complex Event Detection at Wire Speed with FPGAs. *VLDB Endow.*, 3(1-2):660–669, September 2010.

[49] Noa Zilberman, Yury Audzevich, G. Adam Covington, and Andrew W. Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro*, 34(5):32–41, Sept 2014.

[50] Noa Zilberman, Matthew P Grosvenor, Diana Popescu, Neelakandan Manihatty-Bojan, Gianni Antichi, Marcin Wójcik, and Andrew W Moore. Where has my time gone? In *Passive and Active Measurement*, pages 201–214. Springer, March 2017.

[51] Noa Zilberman, Philip M. Watts, Charalampos Rotsos, and Andrew W. Moore. Reconfigurable Network Systems and Software-Defined Networking. *Proceedings of the IEEE*, 103(7):1102–1124, July 2015.

[52] Data Plane Development Kit. `http://dpdk.org/`.

Protego: Cloud-Scale Multitenant IPsec Gateway

Jeongseok Son[†⋆], Yongqiang Xiong[⋆], Kun Tan[‡], Paul Wang[⋆], Ze Gan[⋆], Sue Moon[†]
[⋆]*Microsoft Research,* [†]*KAIST,* [‡]*Huawei*

Abstract

Virtual cloud network services let users have their own private networks in the public cloud. IPsec gateways are growing in importance accordingly as they provide VPN connections for customers to remotely access these private networks. Major cloud providers offer IPsec gateway functions to tenants using virtual machines (VMs) running a software IPsec gateway inside. However, dedicating individual IPsec gateway VMs to each tenant results in significant resource waste due to the strong isolation mechanism of VMs.

In this paper, we design *Protego*, a distributed IPsec gateway service designed for multitenancy. By separating the control plane and the data plane of an IPsec gateway, Protego achieves high availability with active redundancy. Furthermore, Protego elastically scales in and out by seamlessly migrating IPsec tunnels between the data nodes without compromising their throughput. Our evaluation and simulation based on production data show that Protego together with a simple resource provisioning algorithm saves more than 80% of the resources compared with allocating independent VMs.

1 Introduction

Major cloud providers offer virtual networks as a service to customers so that they can setup their own private network topology in the cloud [1, 8, 4]. Tenants create virtual networks and connect applications running inside virtual machines (VMs) to operate their own distributed services. The ease of management, flexibility and elasticity of a virtual network has driven enterprise customers to extend their existing networks using cloud service in lieu of physical network [29].

To seamlessly incorporate remote virtual networks into existing on-premises networks, tenants establish site-to-site VPN connections between the gateways. For site-to-site VPN connections, IPsec is typically used to have secure communication between on-premises and cloud networks. Hence, cloud providers provide tenants with IPsec gateways in addition to the virtual network service. IPsec gateways in on-premise networks peer with them to initiate IPsec tunnels [9].

It is thus crucial for cloud providers to have a flexible and scalable way to provide IPsec gateway functionality to tenants. The current state of the art is shipping software IPsec gateway to tenants using VMs following the trend of Network Function Virtualization (NFV) [29]. Once a tenant makes a request to create an IPsec gateway, an IPsec gateway VM is dedicated to the tenant. It is a natural approach as VMs are basic resource allocation blocks in cloud environments and provide inherent isolation mechanism.

However, dedicating IPsec gateway VMs to tenants results in significant waste of resource for two reasons. First, VMs exclusively occupy a fixed amount of resource. Hence, cloud providers should over-provision the VMs for peak VPN traffic demand. If a tenant does not utilize all the allocated resource of VMs, the unused portion of it is just wasted. Second, each independent gateway VM needs a high availability (HA) setup, which requires additional redundancy. Since VM startup takes several minutes in the cloud due to resource allocation and data copy [38], a passive standby node is typically introduced for fast failover [6]. If every IPsec gateway requires HA, they capture twice as much resource as they actually need.

These limitations have led us to devise a new IPsec gateway architecture to serve multiple tenants with shared resources. To this end, we propose *Protego*, a cloud-scale software IPsec gateway. We design Protego with the following properties: (1) *multitenancy* to serve multiple tenants without violating the bandwidth requirement of each tenant, (2) *elasticity* to seamlessly scale in and out according to the aggregated traffic demand across tenants, and (3) *high availability* to provide reliable service to users without reserving a passive standby for every active VM.

To achieve both high availability and elasticity, Protego separates the control plane from the data plane. For high availability, the relatively long-lived control plane states are saved to a centralized control node. On the other hand, the data plane state is costly to preserve in the same way since it changes every packet sent and received. Hence, Protego saves it locally in data nodes and quickly reconstruct it via the alive control node in case of failure. Protego migrates tunnels between the data nodes without tearing down an old tunnel through rekeying process. This enables Protego to elastically allocate and de-allocate VMs according to varying IPsec traffic of tenants.

Our evaluation using the prototype implementation presents that Protego can migrate IPsec tunnels even without a transient bandwidth degradation. Based on this seamless tunnel migration, we design a provisioning algorithm to autonomously adjust the amount of resource it subscribes. We show that it is possible to save more than 80% of the resources compared with allocating independent VMs to tenants while meeting the bandwidth guarantee to tenants.

To summarize, we make the following contributions: (1) We present a new architecture of distributed IPsec gateway for the cloud which enables high availability with active redundancy. (2) We devise an IPsec tunnel migration scheme that does not compromise the bandwidth of a tunnel during the migration for elastic resource provisioning. (3) We demonstrate Protego with a simple provisioning algorithm indeed saves significant resources through our evaluation and simulation based on production data.

2 Background and Motivation

We first describe why and how IPsec gateways are deployed in cloud environments. Then we identify the necessity of cloud-scale IPsec gateway by showing the resource usage of the IPsec gateways deployed in our data centers. We finally enumerate the requirements of an IPsec gateway for the cloud and challenges of accomplishing it.

2.1 Virtual network and site-to-site VPN

The majority of users who purchase the virtual cloud networks are enterprise customers [29]. They use the virtual network services to extend their on-premises network into the cloud. Since virtual networks provide customers with private IP address space, they can seamlessly move their corporate network to the cloud to take advantage of the flexibility of cloud environments.

To connect a virtual network in the cloud to an existing on-premises network, site-to-site VPN is typically used. Site-to-site VPN remotely connects the entire networks from one another over the public Internet. The VPN connection is established between two VPN gateways. Then they encapsulate outbound traffic and decapsulate inbound traffic rather than individual hosts do so.

2.2 IPsec gateway

IPsec is a de-facto standard for site-to-site VPN connections. IPsec ensures secure communication between the peers by authenticating and encrypting IP packets. For site-to-site VPN, an IPsec gateway encapsulates the entire packet to create a virtual hop, an *IPsec tunnel*, between the peer gateways.

IPsec primarily consists of two protocols: Internet Key Exchange (IKE) and Encapsulating Security Payload (ESP)[1]. The main purpose of IKE is to authenticate the peer and setup the shared attributes between the peers for secure communication. A set of those attributes is called a security association (SA). IKE protocol is used to settle those SAs. ESP protocol encrypts packets to provide confidentiality, integrity and data origin authenticity using negotiated symmetric keys.

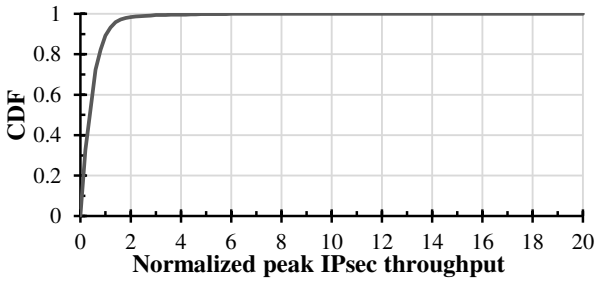


Figure 1: CDF of the peak IPsec throughput of data centers

When an IPsec tunnel is established, initial message exchanges first generate an IKE SA for the peers, which contains a shared key and a cipher suite used to encrypt bidirectional IKE traffic. The shared attributes for ESP encryption and decryption, called CHILD SA, are negotiated securely via further IKE message exchanges. CHILD SAs are unidirectional so the inbound and outbound ESP traffic are encrypted with a different SA.

2.3 Motivation: Inefficient resource usage of IPsec gateway VMs

A prevalent way for cloud providers to deploy IPsec gateways is using VMs running the software implementation of it inside [29]. VMs let them make the best use of their existing commodity server resources and VM management system without installing additional hardware middleboxes. VMs also provide isolated performance for each tenant and can easily scale by dynamically creating or destroying instances.

However, we found that VMs allocated per tunnel underutilizes resources significantly for the following two reasons.

Exclusive resource allocation. Once a VM is allocated to a tenant, the resources of the VM becomes exclusively dedicated to the tenant. Thus, even when a tenant does not fully utilize the capacity of an IPsec gateway, the remaining resources of it cannot be used for serving other tenants' demand.

Figure 1 shows the cumulative distribution of the peak aggregated throughput of all IPsec gateways in each of our data centers. The actual bandwidth values are normalized by the maximum bandwidth that a single IPsec gateway supports. In each data center, there are as many

[1] Authentication Header (AH) is an alternative protocol, but ESP is dominantly used for VPN because only ESP provides confidentiality.

IPsec gateway VMs as there are IPsec tunnels established by tenants. However, the daily peak IPsec bandwidth is less than a single gateway VM capacity in approximately 90% of the data centers. It indicates that most IPsec gateways handle far less traffic than its maximum capacity most of the time.

Even though IPsec gateway VMs have considerable amount of idle resources, there is no easy way to take away the unused resources of VMs for other use. Oversubscribing physical machines with VM consolidation and live migration has been studied as a solution [47, 16, 51]. However, live migration consumes high network bandwidth and easily takes tens of seconds since the whole memory of a VM is iteratively transferred via network [19]. These drawbacks prevent cloud providers from using live migration frequently for flexible resource reallocation.

Passive standby for high availability. IPsec gateways should be highly available since the failure directly results in the downtime of the entire virtual network service. High availability (HA) is generally achieved by using more than one nodes to form a cluster. When one node fails, another node in the cluster quickly takes the role of the failed one. Existing hardware and software IPsec gateways form an active/passive cluster, or 1 + 1 redundancy for HA [12, 2]. The cluster synchronizes the IKE state of an active node with a passive node so that the passive node can keep doing stateful processing after failover.

Although adding a passive standby is a straightforward way to achieve HA, passive backups do not participate in processing IPsec traffic. The resources allocated to passive backups are thus just wasted for HA. In the worse case, 50% of resources is devoted for high availability if every gateway VM has a redundant passive standby.

2.4 Requirements

To overcome the limitations brought up above, Protego should have the following features:

Elastic and scalable capacity adjustment. Protego should be able to save resources without compromising the quality of service of IPsec traffic. It should adjust its capacity by dynamically capturing and releasing resources according to the varying demand of tenants.

High availability with active redundancy. High availability is an essential characteristic to meet service level agreement (SLA). For better resource utilization, Protego should achieve HA with active nodes which process the online traffic rather than with passive standby nodes.

Tunnel performance isolation and guarantee. To make tenants share a single IPsec gateway service, Protego needs to isolate the performance of each IPsec tunnel of tenants so that aggressive users cannot affect the other ones.

2.5 Challenges

A straightforward approach for elasticity and active redundancy is to form a cluster of nodes. Instead of dedicating individual gateway to a tenant, a cloud provider may install the cluster which consists of software or hardware IPsec gateways behind a load balancer and let it process IPsec traffic of multiple tenants. However, the stateful processing of IPsec gateways raises challenges of meeting the requirements in § 2.4 using existing IPsec gateways.

Migrating tunnels without throughput degradation. To elastically adjust the cluster size, a cloud provider should have a means to move the workload between the gateways. A strawman approach is to simply tearing down an existing IPsec tunnel and establish a new one in another gateway. However, this approach leads to significant throughput degradation since the gateways cannot process traffic during the tunnel setup, which requires several sequential round trips of packets. To avoid or alleviate this issue, we should determine how to migrate or share state associated with a tunnel between gateways and when to redirect the packets belong to a tunnel.

Deciding on the right amount of resources to reserve. We need to carefully decide on the amount of resources to reserve due to the latency of spinning up new VMs, which takes several minutes in the major cloud services [38]. Protego would easily violate the performance guarantee until new VMs are added, if it reserves too little resources to save them. On the other hand, it would waste resources if it subscribes too much. Therefore, we should devise a way to determine the proper amount of resources to subscribe in order to save resources while meeting the bandwidth requirement of tenants.

Optimizing the packet processing performance. IPsec packet processing is computationally intensive since it involves encryption and decryption of the payload. To maximize the throughput of Protego, it is crucial to parallelize packet processing using multiple cores. However, IPsec gateways maintain ESP packet counters to include a sequence number in ESP packets for the anti-replay feature [33]. In order to ensure that the sequence number is not reused, a simple method is to make packet processing threads share a global packet counter for each tunnel and update it every packet sent. This approach requires locking, however, which decreases parallelism in packet processing significantly. Hence, it is unsuitable to achieve multiple Gbps per-tunnel throughput we aim to offer.

3 Protego Core Ideas

Protego meets the requirements of a cloud-scale IPsec gateway described in the previous section based on the

following key ideas.

3.1 Separation of control and data planes

In traditional software IPsec gateway implementations, IKE, ESP modules and pertinent state are consolidated into a single node. Each member of IPsec gateway cluster thus has separate IKE module and state.

We propose a separate control node which incorporates the signaling plane of gateways into a single node. The control node deals with traffic steering and dynamic provisioning of the data plane. The data plane of Protego consists of a cluster of VMs and focus on packet encryption and decryption process. By this separation, each plane manages its state to deal with different access patterns and focuses on ensuring different properties.

Keeping control plane state in a central node. Recovering IKE state is costly since re-negotiating an IKE SA takes several sequential round trip of messages [32]. On the other hand, it is updated infrequently, every tens of seconds or every couple of minutes, when it receives heartbeat messages from a peer gateway.

Protego saves control plane state to a centralized control node exploiting this relaxed update frequency. By saving the state to a centralized store every time it is updated, Protego achieves tunnel migration without stopping processing traffic.

Quick recovery of data plane state. ESP data nodes handle data packets to encrypt and decrypt them. An ESP packet counter is updated per-packet basis, which makes it infeasible to store ESP state separately as Protego does for IKE state. However, ESP SA can be initiated in 1 RTT if the IKE SA is alive. Hence, The Protego control plane just re-negotiates the ESP state during failover.

3.2 Seamless tunnel migration by rekeying

The key enabler of elasticity is seamless migration of workload. Protego is able to migrate existing IPsec tunnels from one ESP node to another one leveraging rekeying process of IPsec [32] without impairing their throughput.

IPsec gateways use keys for a limited amount of time. Before a SA expires, a gateway negotiates a new key with its peer. This process is referred to as rekeying. Rekeying is done in parallel without collapsing the old SA. Because Protego has a global signaling plane, it can insert a new key to any data node which will receive a migrated tunnel. Protego seamlessly steers the traffic using software load balancers tailored to IPsec protocol.

3.3 Elastic provisioning algorithm

VM live migration requires operators to apply complex modeling and prediction techniques [36, 15, 50, 17, 49] to minimize the high overhead of live migration. In contrast, we devise a straightforward resource provisioning algorithm leveraging the light-weight and instant migration scheme of Protego. We model the IPsec tunnel placement as a one-dimensional bin packing problem. Solving this problem is not sufficient, however, since we still have to consider the long latency of spinning up VMs. To precisely estimate the amount of resources to subscribe in advance, Protego keeps track of the resource usage distribution of IPsec tunnels and calculate the convolution of these distributions. We will describe those algorithms together in detail in § 5.

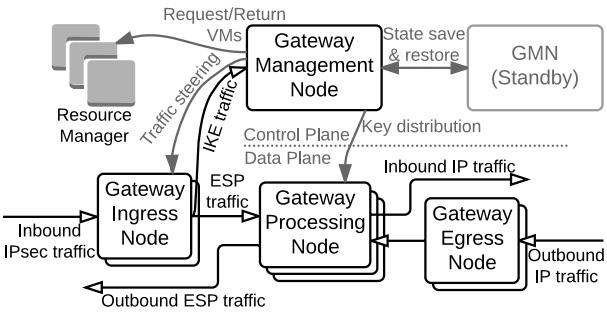


Figure 2: Protego architecture and data flow overview

4 System Design

We present how we design Protego with the core ideas in § 3 to satisfy the requirements enumerated in § 2.4

4.1 Architecture Overview

Protego has separate control plane and data plane. The control plane consists of Gateway Management Node (GMN), a controller which handles IKE traffic and decides the amount of resources to reserve. It also steers ESP traffic by inserting forwarding rules. The data plane consists of a set of Gateway Processing Node (GPN) which processes ESP packets. Gateway Ingress Node (GIN) or Gateway Egress Node (GEN), which are software load balancers tailored for Protego, exposes external virtual IP addresses (VIP) and forward the traffic destined to VIPs to an appropriate node. It also limits the bandwidth of each tunnel for performance isolation. Figure 2 shows the overall architecture of Protego.

4.2 Control Plane: Gateway Management Node

IKE packet processing. GMN processes the IKE traffic of IPsec tunnels. As we discussed above, the main role of IKE is to negotiate SAs that include a cipher suite, and materials to generate symmetric keys with its peer gateway. We do not elaborate on the protocol details, which can be found at RFC5996 [32].

Once a shared symmetric key for ESP encryption is created, GMN distributes this key to one of the nodes in the data plane. Then it adds a rewrite rule to a GIN(Gateway Ingress Node) and GEN(Gateway Egress

Node) to steer the corresponding ESP traffic to a GPN.

Whenever GMN processes a packet, it saves updated IKE SAs to the standby GMN. In case of failure, the standby node takes over the role of the active GMN node. GIN is responsible for detecting the failure of GMN by monitoring heartbeat messages and steering IKE packets to the standby node after a failover.

Resource management. Another important role of GMN is adjusting the number of GPNs in the data plane. When the traffic increases, GMN adds more VMs to the ESP node pool and move some existing tunnels to the new ESP node and vice versa. GMN monitors the CPU utilization of every GPN periodically. When a GPN sends a tunnel migration request to balance the load, GMN selects an appropriate node which can receive the tunnels. If there is no nodes that can receive the tunnels, GMN requests additional VMs to the resource manager of a cloud provider. All request and response packets for the resource management are sent and received using TCP for reliable transmission.

Traffic steering. When an IPsec tunnel is migrated, GMN inserts appropriate forwarding rules to a GEN and a GIN. This process includes the selection of a GPN which receives an IPsec tunnel to be migrated.

4.3 Gateway Ingress and Egress Node

GIN and GEN are analogous to software load balancers, but provide additional features necessary to Protego. We added the following functionalities to Ananta [42], which is a scalable software load balancer with high availability.

Traffic forwarding. The major role of GIN and GEN in Protego is directing packets. They rewrite the destination address to the address of a GPN which is selected to process the traffic. GIN exposes an external VIP which the inbound traffic is destined to. For the inbound traffic, GIN should be able to distinguish different tunnel traffic destined to the same IP in order to distribute ESP packets across different GPNs. GIN matches the Security Parameter Index (SPI) of ESP packets for this. For the outbound traffic, GEN simply uses the traffic selector, an ACL (Access Control List)-like filter exchanged when GMN negotiates CHILD SAs.

Rate limiting. Another important role of GINs and GENs is limiting the bandwidth of tunnels. One of the requirements of Protego is enforcing per-tunnel performance isolation. Protego achieves this by limiting the rate of tunnels to the maximum bandwidth that cloud provider promise to support to tenants.

GPN failure detection. As long as GMN is alive, a peer gateway cannot detect the failure of GPN since GMN keeps transmitting IKE heartbeat messages. GIN and GEN are responsible for detecting the failures. Introducing heartbeat messages is a common technique for this. However, the heartbeat messages with a tiny interval overload the internal network and the detector as the number of nodes grow. Instead of the fixed interval, we want the heartbeat interval of GPNs with higher throughput to be shorter to detect failure more quickly. To do so, GIN/GEN uniformly sample and tag packets to trigger heartbeat messages from GPNs. We describe this process in more detail in § 6.

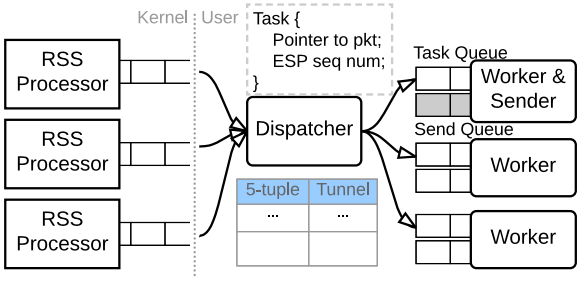


Figure 3: GPN design and packet processing flow

4.4 Data Plane: Gateway Processing Node

GPNs handle encryption and decryption of all tunnels. GMN decides on the mapping between tunnels and GPNs, and inserts forwarding rules to GIN/GEN accordingly. Each GPN also monitors and reports its resource utilization (CPU, bandwidth, etc.) to GMN periodically. When the utilization exceeds a certain threshold, it sends a tunnel migration request to GMN to change the mapping for load balancing.

To optimize the performance of Protego while guaranteeing the uniqueness of sequence number, we avoid using locks with the design depicted in Figure 3. We pin a *worker* thread to each core for all packet processing tasks and make those worker threads run independently from one another. One special worker thread, *dispatcher*, enforces packet ordering within a tunnel and distributes packet processing tasks across multiple cores. Another special type of worker thread, *sender*, is responsible for sending processed packets in batch.

Note that the dispatcher and sender are also worker threads. They are not completely dedicated to the task dispatching and sending. When all the task queue of other workers are full, the dispatcher puts the task to its own queue and performs encryption or decryption. A worker thread becomes a sender only when its send queue has some enqueued send requests. This design choice is for maximizing encryption and decryption performance by fully utilizing CPU cores under heavy workloads.

4.5 Tunnel migration

IPsec tunnel migration is an essential operation for elasticity. Protego leverages rekeying process of IKE to mi-

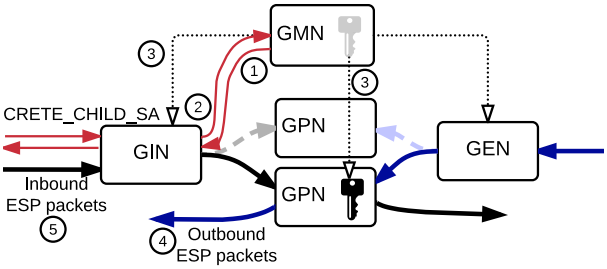


Figure 4: Tunnel migration process

grate a tunnel from one GPN to another one. Following is the detailed tunnel migration steps depicted in Figure 4.

1. GMN sends the CREATE_CHILD_SA request with a new Diffie-Hellman (DH) value[2] and a nonce.
2. GMN receives the CREATE_CHILD_SA response which include the DH value and the nonce of a responder. GPN generates two new child SAs using those information for the inbound and outbound tunnels.
3. GMN hands the new SAs over to a GPN which would receive the tunnel to migrate. GMN also adds a corresponding steering rule to GIN and GEN using the SPI and the traffic selector of new SAs known by the CREATE_CHILD_SA exchange.
4. GPN starts to use the new outbound SA. Once the peer gateway receives this traffic of new inbound SA, it starts to use its new outbound SA.
5. GIN steers the ESP packets destined to new inbound SA of Protego to the new GPN. The old inbound SA is no longer used.

The old SAs are not destructed during the migration process, so Protego can seamlessly migrate tunnels without affecting the performance.

5 Elastic Resource Provisioning

We present an algorithm to dynamically provision and de-provision the data plane.

5.1 Objectives

Our algorithm has two conflicting goals. One is to minimize the resource usage for better efficiency, and the other one is satisfying the throughput requirement of tenants.

Therefore, it is critical for Protego to gauge the minimum amount of resources, or the number of VMs needed to reserve to ensure the per-tunnel performance to tenants. We precisely model the resource requirements and use a bin packing algorithm to figure it out.

[2]The Diffie-Hellman value can be excluded complying with the IKEv2 specification. We added it just for stronger guarantees of forward secrecy.

Notation	Explanation
σ_i	Maximum CPU usage of a tunnel i
α_i	Current CPU usage of a tunnel i
β_j	Current CPU utilization of a node j
U_i	Probability distribution of the CPU utilization of a tunnel i
ε	Throughput guarantee violation tolerance
Y	Probability distribution of aggregated tunnel CPU utilization
C	Number of VMs reserved for Protego
T_H	CPU utilization threshold for hotspot detection

Table 1: Variables used in the algorithm description

5.2 Model

Hierarchy of virtual machine. VM states are classified into roughly three categories in the cloud. *Active VMs* are booted VMs actively used by a service. *Shutdown VMs* are not yet booted and not under control of any one. In addition to those typical states, *Inactive VMs* [45] are booted and under control of a service but reserved for scaling out the service capacity. By introducing the inactive state, cloud providers are allowed to reduce the resources allocated for those inactive VMs.

Each VM group has a different latency to be added to the ESP node pool. Normally, active and inactive VMs are added almost instantly since they are controlled by a service, but shutdown VMs take at least several minutes to be active. If the service does not reserve enough active and inactive VMs, tenants may experience severe performance issue.

Node capacity. In IPsec gateways, the CPU resource of nodes is the bottleneck that determines the throughput of IPsec tunnels. We assume that every node has the same CPU resource, and regard all nodes have normalized CPU capacity 1.

Maximum resource usage. The maximum CPU usage of tunnels is bounded in our case since we limit the bandwidth of tunnels. We express the maximum limit of the tunnel CPU utilization as a real number σ_i. $(0 < \sigma_i < 1)$

Current resource usage of a tunnel and utilization of a node. The CPU usage of a tunnel is periodically calculated with the interval of τ. Let α_i is the current CPU usage of a tunnel at specific times. Then the CPU utilization of a node is defined as $\beta_j = \sum_{i=1}^{k} \alpha_i$, where k is the number of tunnels in the node.

Resource usage distribution of a tunnel. The CPU usage of a tunnel varies over time. The usage distribution of a tunnel U_i takes this into account. $U_i(x)(0 < x < \alpha_i)$ is the probability density function of the CPU usage, which

shows the likelihood of how much CPU resource a tunnel would consume at a certain time.

Violation tolerance. The violation tolerance ε expresses how tolerable the system is on the throughput guarantee violation. If the traffic of a tunnel during a certain time interval is not fully served due to the insufficient resources of a GPN, the tunnel fails to achieve demanded throughput of a tenant as packets get dropped. The sum of the time intervals of such time should account for less than ε of the total available time of Protego.

5.3 Minimum number of VMs for per-tunnel throughput guarantee

Based on the model described in § 5.2, we figure out the minimum number of VMs that Protego should reserve to satisfy the IPsec tunnel throughput guarantee to tenants.

Aggregated traffic distribution. Let $Y = \sum_{i=1}^{n} U_i$, where n is the number of all tunnels. Y denotes the probability distribution of aggregated CPU usage of all tunnels in the system. Since U is a discrete probability distribution, we can calculate the convolution of any two resource usage distributions using the following formula.

$$Y(z) = \sum_{k=0}^{1} U(k)U(z-k)$$

We use the formula to sum up n resource usage distributions inductively to the Y. We assume that the tunnel resource usage distributions are independent from one another.

Minimum number of VMs for the throughput guarantee. The throughput guarantee constraint is formally expressed with Y and ε:

$$Pr(Y > C) \leq \varepsilon$$

where C is the total resource of the system. In our case, C is the number of active and inactive VMs because all VMs have the normalized CPU capacity 1. ε is a given constant and Y is derived from U_i. Hence, we can figure out C, the number of VMs that Protego needs to reserve to guarantee the throughput.

Protego should keep its number of active and inactive VMs above C so that the probability of the violation is maintained below ε. In a real deployment, however, we need to take the T_H into account since it incurs a small resource waste. Thus, the number of VMs it reserves should be higher than C/T_H. We assume the degree of external fragmentation of the capacity of GPNs is negligible here.

5.4 Load balancing and tunnel consolidation

Protego detects nodes which the demand of assigned tunnels exceeds its capacity and balance the workload by migrating the tunnels to other relatively idle nodes. At the same time, Protego periodically consolidates tunnels to minimize the number of active VMs.

Hotspot node detection. GMN should detect nodes of which the demand of tunnels exceeds its capacity. We set a CPU utilization threshold $T_H > max\ \alpha_i$ and regard a node as hotspot if $\beta_j > T_H$. T_H should be large enough to ensure high utilization of nodes.

Tunnel migration. Once the hotspot node is detected, a subset of the tunnels in the node should be migrated to lower β_j below T_H. To minimize the number of migration, the tunnels are sorted in decreasing order of α_i, and largest k tunnels where $\sum_{i=1}^{k} \alpha_i > \beta_j - T_H$ are chosen and migrated in that order. The same Best Fit algorithm is used to choose a node to place each tunnel. The system adds an inactive VM to the active pool if none of the nodes are not able to receive the tunnel.

Tunnel consolidation. Protego periodically decides new tunnel allocation based on Best Fit Decreasing (BFD) algorithm, which guarantees to use no more than $11/9$ bins of the optimal solution [25]. Protego periodically sorts all tunnels in decreasing order of α_i, and use BFD to figure out a new placement of the tunnels. After every consolidation, Protego makes empty active VMs inactive to minimize the number of active VMs.

6 Implementation

6.1 GIN & GEN

GIN and GEN are both based on our packet filtering driver based on Windows NDIS Lightweight filter (LWS) driver. The main task of GIN and GEN are modifying the destination IP address of the packets to forward them to a right GPN which possesses the shared keys for the inbound and outbound traffic of a tunnel that packets belong to. For this purpose, GIN and GEN maintain the mappings between SPI and GPN IP addresses, and traffic selectors and GPN IP addresses.

Another important role of GIN and GEN is detecting the failure of GPNs. GIN and GEN manipulate the last bit of TOS field in outer IP header of ESP packets for tagging. They sample a part of packets and set the last bit of TOS to 1. Once a GPN detects the bit is set, it mirrors the packet with the reversed source and destination addresses and empty payload back to GIN/GEN as a heartbeat. When there is no reply within a certain period, the packet is regarded as dropped. After three consecutive drops, GIN or GEN concludes that a corresponding GPN fails.

6.2 GMN

We implement GMN based on the existing IPsec service module in the Routing and Remote Access Service

(RRAS) [14]. We add the state backup and recovery logic to the implementation of Remote Access service in Windows Server 2012 R2. Our modified RRAS captures state modification by wrapping global variables with setter functions. Also, public interface is added to expose states and save the changed ones to an external IKE module of the passive GMN. These interfaces are implemented based on asynchronous RPC (Remote Procedure Call) already implemented in Windows Server [10].

6.3 GPN

We implement our own filter driver to catch packet receive notifications from NIC and return the address of a free buffer in the Free Buffer Queue. NIC copies received packet data to the buffer, then the filter driver pushes the pointer to the receive queue exposed to user space.

Dispatcher thread maintains an array of buffers to hold the encrypted or decrypted packets to be sent in a batch. It pushes the request to one of the Task Queue of worker threads and a worker encrypts or decrypts the packet data in turn. The worker writes back the process packet to the array of buffer at the assigned index, and increases the processing counter. Once the processing counter reaches the total size of the array size, the sender thread starts to send out the whole buffer. Upon receiving the send completion notification, the buffers are returned to Free Buffer Queue maintained by our filter driver so that it can be reused.

Note that GIN, GEN, GMN and GPN can be implemented independently on top of different platforms although we implemented all of them in Windows servers in our local test bed. They can be built and combined on public clouds by third-party as well if an enterprise tenant wants to deploy their own VPN service.

7 Evaluation

The test bed has the same networking and configuration as our real production IPsec gateway environment. The experimental setup consists of 32 servers with 16-core Intel Xeon E5-2650 v2 CPU working at 2.6Ghz and Mellanox Connect-3 Pro 40Gbps NIC. We use Windows Server 2012 R2 and Hyper-V.

Figure 5 shows the topology of the experiment. We use a WAN emulator to emulate latency and packet loss.

7.1 Failover

To evaluate the impact of failures in Protego, we establish an IPsec tunnel between Protego and the IPsec gateway in the user network in our experimental topology. The client sends 300 Mbps of TCP traffic to the server machine. While the client is sending the traffic, we power off GPN and GMN one by one and monitor the throughput at the server side. We set the sampling

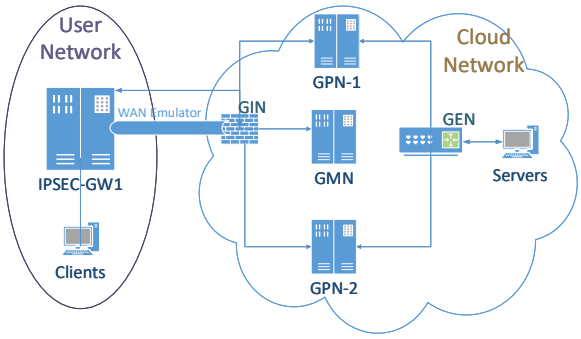


Figure 5: Experiment topology

rate of GIN and GPN for failure detection to 1/1000 and the minimum sampling interval to 10 ms.

We powered off GPN at around 18 second. In Figure 6, the throughput drops slightly as some packets are dropped during the failover period. Once a new ESP key is negotiated and inserted to a new GPN, the throughput recovers to the original value after the TCP slow-start phase. In the GMN failure case, the throughput of the tunnel is not degraded as shown in Figure 6, since CHILD SAs are alive and used for ESP packet processing, and GMN is restored almost instantly.

Figure 8 shows the latency of failover and IKE state update, which we measured running the operations 20 times. It takes 0.28 seconds in total for the failover. The round trip time between peer gateways for re-negotiating a new CHILD SA accounts for 68% of the total failover time. The latency of updating an IKE SA in a passive GMN is 89 ms, which is quick enough to handle IKE heartbeat messages sent every few seconds.

7.2 Tunnel migration overhead

We created two GPNs as described in Figure 5 to see the throughput change of an IPsec tunnel during migration. We measured the throughput of a TCP stream in the server.

Figure 7 shows the throughput of the IPsec tunnel over time. We exposed the tunnel migration API to manually initiate the process via command line of GMN. The migration process is started at approximately 18 seconds. The tunnel performance is maintained during the migration process according to the figure. The time it takes to migrate a tunnel is the same as the sum of the rekey and ESP state insertion time mentioned in the failover section.

7.3 GPN performance

Multi-core throughput. In order to measure the performance and multi-core scalability of GPN, we establish a single IPsec tunnel between the IPsec gateway in the user network and one of the GPNs of Protego. To measure the encapsulation performance, the server sends

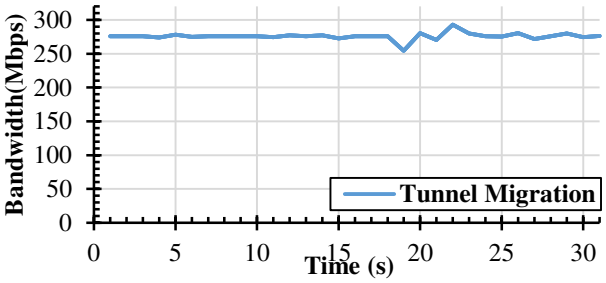


Figure 6: Impact of failure on tunnel throughput

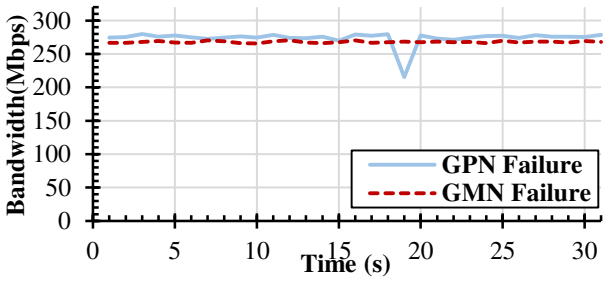


Figure 7: Impact of migration on tunnel throughput

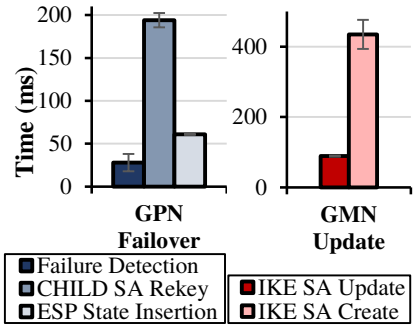


Figure 8: Failover and state update latency breakdown

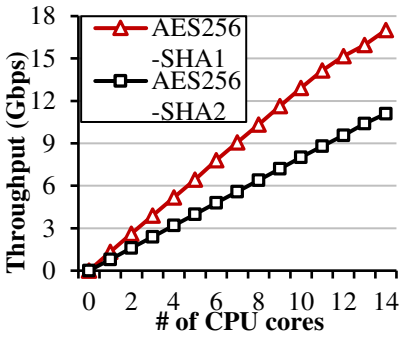


Figure 9: Single node throughput

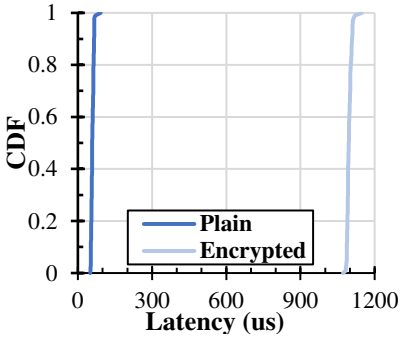


Figure 10: CDF of latency per packet

TCP traffic to a client. The TCP packet length is 1400 bytes. We used a number of TCP connections to fully saturate the CPU resource of the GPN.

Figure 9 shows the throughput of an IPsec tunnel measured in the server using the aggregated TCP throughput. As the number of CPU cores increases, the throughput of a single tunnel performance of a single GPN increases linearly. Protego can provide 10 Gbps of the throughput with 8 cores when AES256-CBC is used for encryption and SHA1 is used for integrity. When SHA2 is used for integrity, more than 12 cores is required to achieve 10 Gbps in our evaluation setup.

Packet processing latency. We also measured latency added by GPN node. To quantify the latency incurred by a GPN node, we measure the latency of packets which only pass through GIN and skip GPN, and then that of packets processed by a GPN to encrypt them with AES256CBC-SHA1. A client sends 1400 bytes TCP packets of which payload contains timestamp value. A server which receives the packet prints out the latency based on the embedded timestamp. We turned off WAN emulator in this evaluation and place all VMs in the same rack.

We sampled 1,000 packets to draw CDF graph in Figure 10. The deviation of latency distribution is quite small as they are connected by a single ToR. The median value of the case when GPN is not involved is 61 us, and is 1094 us when GPN is involved. The latency overhead of Protego is around 1 ms. It is negligible compared with RTT of WAN, which is tens or hundreds of ms in general.

7.4 Resource provisioning simulation

We evaluate the algorithm elaborated in § 5 by doing a large-scale simulation. We use the throughput data of IPsec tunnels in our data centers to figure out how much resource is saved by Protego compared with the existing VM allocation based system. We collected the hourly average throughput of IPsec tunnels for 24 hours. We divide the actual tunnel throughput values by the maximum capacity of deployed IPsec gateways. Then we multiply the resulting ratio by an arbitrary maximum tunnel throughput we choose for simulation.

Resource saving. We collected the 1-day throughput data of IPsec gateways in one of our data centers. The average throughput of the tunnels is measured every minute. We assume that all GPNs have the same processing capacity, and all ESP packets with the same size consume the same amount of CPU resource when processed.

The throughput trace of 170 tunnels was collected and used in our simulation. We normalized the maximum tunnel throughput to 1.5 Gbps, which is the maximum tunnel throughput supported by major cloud providers [3, 5]. The GPN capacity is set to 5 Gbps. ($\sigma_i = 0.3$) The hotspot threshold $T_H = 0.90$ and the throughput measurement interval is 1 minute. Also, the violation tolerance $\varepsilon = 0.95$.

Figure 11 displays illustrates the aggregated IPsec throughput of all tunnels and the total capacity of active

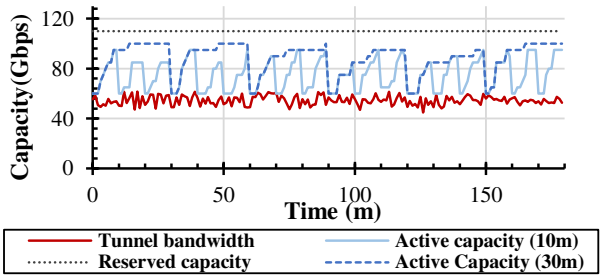


Figure 11: Resource provisioning efficiency

VMs used as GPNs. The reserved capacity represents the total resource of all VMs that Protego reserves by figuring out the minimum number of VMs it needs for bandwidth guarantee based on the formula explained in § 5.3. In this simulation, the reserved capacity is 110 Gbps since Protego subscribes 22 VMs.

The consolidation interval is set to 10 minutes and 30 minutes respectively. The number of active VMs grows between the consolidation points since Protego balances the IPsec workload by migrating the tunnels as their throughput are fluctuating. The number of active VMs shrinks every consolidation interval.

It is trivial from the figure that the smaller the consolidation interval is, the less active VMs the Protego utilizes. The average provisioned capacity of active VMs is 65.38, 74.75 Gbps, and 88.17 Gbps for the 5-minute, 10-minute, and 30-minute consolidation intervals respectively. The average total throughput of the tunnels is 57.49 Gbps. The trade off of finer consolidation interval is investigated using the result in the next subsection.

Throughput guarantee. Another important requirement of a resource provisioning algorithm is to meet the throughput guarantee. We introduce *daily bandwidth guarantee* to measure how much time Protego actually provisions enough resources in a similar way as availability SLAs are defined.

$$DailyBandwidthGuarantee(\%) = \frac{TotalAvailableMinutes - MinutesOfViolation}{TotalAvailableMinutes}$$

The violation happens when the sum of the demand bandwidths of IPsec tunnels, which are rate limited, exceeds the capacity of a GPN. We assume that the packet scheduler of GPNs is completely fair so the bandwidth guarantee is violated only when its bandwidth demand is larger than its fair share. In public clouds, only the availability of VPN services are guaranteed [7, 13]. Cloud providers seldom guarantee the bandwidth in the SLA in any form [39]. We suggest the bandwidth guarantee to briefly show the trade-off between the utilization and the QoS with different consolidation intervals. We do not determine the optimal parameters of our algorithm here, which will be different depending on the internal performance indicators of each cloud provider.

Consolidation Intervals	3 min	5 min	10 min	30 min	60 min
Active VM Capacity (Gbps)	61.23	66.17	73.97	88.34	93.22
99th-percentile Guarantee (%)	90.21	93.07	96.84	98.24	98.63
Resource Saving (%)	88.00	87.03	85.50	82.68	81.72

Table 2: Bandwidth guarantee and resource saving achieved with different consolidation intervals

Table 2 contains the detailed numbers we get from the simulation. The resource saving is calculated by dividing the capacity of active VMs by the total capacity of VM assuming that one VM is dedicated to each tunnel. Since there are 170 tunnels of which maximum bandwidth is 1.5 Gbps, the total capacity is 255 Gbps to provision for peak demands. Moreover, the high availability requirement doubles the number of necessary VMs. Therefore we figure out the total capacity required for the old system is 510 Gbps. When the consolidation interval is 10 minutes, Protego can save around 85.50% of VM resource while meeting the bandwidth guarantee of 99 % of the tunnels for 96.84 % of the total available time of Protego.

8 Discussion

Security implication. One may argue that the security of the overall system is weakened due to the risk of placing secret keys in a shared VM, GMN. However, the VMs of Protego are not leased to tenants but are under control of cloud providers. They can block external network access to those control nodes as they normally do for their internal servers. Note that Protego performs complete IPsec protocol as it is. Rekeying process for migration may incur some overhead but does not compromise security.

Keeping occasionally changed state in a centralized node. We make Protego keep the IKE state in a centralized node. Likewise, the same approach could be applied to other NFs to make the data plane stateless. For example, asset monitoring systems such as PRADS [11] employ fingerprints to identify clients. Since they are rarely changed, storing them in a centralized node would be a good way to build a scalable monitoring system.

9 Related Work

Software NFs for the cloud. Flexible and easy to manage software NFs are becoming more prevalent in data centers these days [42, 21, 23, 24, 22]. Especially, software load balancers are deployed and replacing hardware ones. Ananta [42] is the first software load balancer specially designed for cloud environments. Ananta has a

separate control plane and data plane. Yoda [23] decouples the flow state from load balancers and stores it in a persistent storage for high availability. Ananta and Yoda have influenced the design of Protego. Maglev [21] is a software load balancer, further optimized for the throughput of a single machine. Maglev employs a forwarder thread which calculates the 5-tuple hash of the packets and put them into the receiving queue of a dedicated packet rewriter thread. The dispatcher thread of Protego plays a similar role to the forwarder and steering thread. However, the data plane design of Protego is different from Maglev and other packet processing frameworks [20, 30, 34, 35, 28] in that it is specially designed for IPsec. Protego takes the state dependency between ESP packets into account and enables even the packets belong to the same tunnel distributed across multiple worker threads.

NFV Frameworks for scalability and availability. OpenNF [27] controller manages both the forwarding rules of SDN controller and the internal state of NFs to migrate flows from one NF instance to another. OpenNF controller buffers the packets of the flow in migration until the corresponding per-flow state is moved, which adds hundreds of milliseconds of per-packet latency. U-HAUL [37] selectively apply the OpenNF migration scheme to elephant flows to optimize the migration performance. Unlike these controllers, Protego achieves loss-free migration without migrating the per-tunnel state by leveraging the rekeying feature of IKE protocol.

E2 [40], Stratos [26], and OpenBox [18] are frameworks that provide high-level means of developing, placing and scaling NFs by introducing an NF-agnostic controller. We want to point out that employing an NF-specific controller is often necessary and efficient as we have shown. StatelessNF [31] uses a low-latency data store to make NFs stateless for scalability and high availability. Protego has a similarity to StatelessNF in that it stores state in a centralized node, but Protego maintains the frequently changing state locally. NetBricks [41] is a framework built on Rust to ensure safe memory isolation in user-level for NFs without VMs. Unlike NetBricks, we still use VMs for cloud providers to leverage existing resource allocation and management system based on virtualization platforms. FTMB [44] and Pico Replication [43] leverage VM checkpoint or snapshot to ensure high availability of middleboxes. We avoid checkpoint-based approaches since VM restore time in the cloud easily takes several minutes [38], which is too long to meet the tight availability SLA of cloud providers.

We want to emphasize that the efficiency improvement has been achieved by taking multitenancy into account when designing our system. The NFs for cloud environments should have a means to seamlessly and quickly migrate workloads and resource allocation/deallocation policy to elastically adapt to varying demand of tenants.

Resource provisioning in shared environment. Deciding the right amount of resources to provision in a shared environment has been a long-standing problem. Urgaonkar et al. [46] shows the gain of oversubscribing the resource in a shared hosting platform. They profile the resource usage of applications offline with realistic workloads. Based on those resource usage distributions, the system decides the capacity of resources. We have adopted the resource usage model of them to decide the minimum number of VMs to reserve.

In the context of cloud computing, VM placement and migration is one of the most widely studied areas. Sandpiper [49] leverages VM live migration to balance the workload of overloaded physical machines. Bobroff et al. [17] takes SLA into account and design a forecasting technique to minimize SLA violation. Verma et al. [47] considers dynamic VM resizing to efficiently consolidate VMs with less frequent migration. Though there has been a large volume of literature along this line, the high network bandwidth consumption of live migration and long migration time of hotspot VMs [48] hinder the wide deployment of it. Protego provides a migration scheme which is far more lightweight and quicker than VM live migration. In addition, it provides the finer granularity of load balancing by migrating tunnels instead of an entire VM. Therefore, it has a better potential for real production use.

10 Conclusion

We have described Protego, a software IPsec gateway specifically designed for cloud environments. Protego serves multiple tenants using shared resources for statistical multiplexing. It separates the control plane from existing IPsec gateways and preserves its state for high availability. We leverage IKE rekeying feature to seamlessly migrate tunnels without impairing their throughput. We devise a resource provisioning algorithm and demonstrate that Protego can save more than 80% of the resources comparing with existing approach, while guaranteeing the IPsec throughput for higher than 90% of uptime.

Acknowledgements

We thank our shepherd Ittay Eyal and the anonymous reviewers for their helpful feedback. We are also grateful to Shinae Woo and Bojie Li for their thoughtful comments on the drafts, and Joongi Kim for providing his figure templates. This work was supported in part by the MISP (Ministry of Science, ICT & Future Planning), Korea, under the National Program for Excellence in SW (2016-0-00018) supervised by the IITP (Institute for Information & communications Technology Promotion) (2016-0-00018).

References

[1] Amazon Web Service - Virtual Private Cloud (VPC). https://aws.amazon.com/vpc/.

[2] Cisco - IPSec Stateful Failover (VPN High Availability) Feature Module. http://www.cisco.com/c/en/us/td/docs/ios/12_2/12_2y/12_2yx11/feature/guide/ft_vpnha.html.

[3] Google - VPN throughput. https://cloud.google.com/compute/docs/vpn/advanced.

[4] Google Cloud Platform - Virtual Network. https://cloud.google.com/virtual-network/.

[5] Microsoft Azure - About VPN Gateway. https://docs.microsoft.com/en-us/azure/vpn-gateway/vpn-gateway-about-vpngateways.

[6] Microsoft Azure - Planning and design for VPN Gateway. https://azure.microsoft.com/en-us/documentation/articles/vpn-gateway-plan-design/.

[7] Microsoft Azure - SLA for VPN Gateway. https://azure.microsoft.com/en-us/support/legal/sla/vpn-gateway/.

[8] Microsoft Azure - Virtual Network. https://azure.microsoft.com/en-us/services/virtual-network.

[9] Microsoft Azure - VPN Gateway. https://azure.microsoft.com/en-us/services/vpn-gateway/.

[10] Microsoft Remote Procedure Call. https://msdn.microsoft.com/en-us/library/windows/desktop/aa378651(v=vs.85).aspx.

[11] PRADS - Passive Real-time Asset Detection System.

[12] strongSwan - High Availability. https://wiki.strongswan.org/projects/strongswan/wiki/HighAvailability.

[13] VPN Service Level Agreement (SLA). https://cloud.google.com/vpn/sla.

[14] Windows Remote Access Service. https://technet.microsoft.com/en-us/library/dn636119(v=ws.11).aspx.

[15] S. Akoush, R. Sohan, A. Rice, A. W. Moore, and A. Hopper. Predicting the Performance of Virtual Machine Migration. In *MASCOTS*, 2010. IEEE.

[16] S. A. Baset, L. Wang, and C. Tang. Towards an Understanding of Oversubscription in Cloud. In *HotICE*, 2012. USENIX.

[17] N. Bobroff, A. Kochut, and K. Beaty. Dynamic Placement of Virtual Machines for Managing SLA Violations. In *INM*, 2007. IEEE.

[18] A. Bremler-Barr, Y. Harchol, and D. Hay. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *SIGCOMM*, 2016. ACM.

[19] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *NSDI*, 2005. USENIX.

[20] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *SOSP*, 2009. ACM.

[21] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *NSDI*, 2016. USENIX.

[22] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey. Bohatei: Flexible and Elastic DDoS Defense. In *USENIX Security*, 2015. USENIX.

[23] R. Gandhi, Y. C. Hu, and M. Zhang. Yoda: A Highly Available Layer-7 Load Balancer. In *EuroSys*, 2016. ACM.

[24] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud Scale Load Balancing with Hardware and Software. In *SIGCOMM*, 2014. ACM.

[25] M. R. Garey, R. L. Graham, and J. D. Ullman. Worst-case Analysis of Memory Allocation Algorithms. In *STOC*, 1972. ACM.

[26] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar. Stratos: A Network-Aware Orchestration Layer for Middleboxes in the Cloud. In *CoRR*, 2013.

[27] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling Innovation in Network Function Control. In *SIGCOMM*, 2014. ACM.

[28] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang, and K. Park. APUNet: Revitalizing GPU as Packet Processing Accelerator. In *NSDI*, 2017. USENIX.

[29] A. Greenberg. Windows Azure: Scaling SDN in the Public Cloud. Open Networking Summit (ONS) 2014.

[30] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated Software Router. In *SIGCOMM*, 2010. ACM.

[31] M. Kablan, A. Alsudais, E. Keller, and F. Le. Stateless network functions: Breaking the tight coupling of state and processing. In *NSDI*, 2017. USENIX.

[32] C. Kaufman, P. Hoffman, Y. Nir, and P. Eronen. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 5996, RFC Editor, 2010.

[33] S. Kent. IP Encapsulating Security Payload (ESP). RFC 4303, RFC Editor, 2005. `http://www.rfc-editor.org/rfc/rfc4303.txt`.

[34] J. Kim, K. Jang, K. Lee, S. Ma, J. Shim, and S. Moon. NBA (Network Balancing Act): A High-performance Packet Processing Framework for Heterogeneous Processors. In *EuroSys*, 2015. ACM.

[35] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *SIGCOMM*, 2016. ACM.

[36] H. Liu, C.-Z. Xu, H. Jin, J. Gong, and X. Liao. Performance and Energy Modeling for Live Migration of Virtual Machines. In *HPDC*, 2011. ACM.

[37] L. Liu, H. Xu, Z. Niu, P. Wang, and D. Han. U-HAUL: Efficient State Migration in NFV. In *APSys*, 2016. ACM.

[38] M. Mao and M. Humphrey. A Performance Study on the VM Startup Time in the Cloud. In *CLOUD*, 2012. IEEE.

[39] J. C. Mogul and L. Popa. What We Talk About when We Talk About Cloud Network Performance. In *SIGCOMM CCR*, 2012. ACM.

[40] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A Framework for NFV Applications. In *SOSP*, 2015. ACM.

[41] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. NetBricks: Taking the V out of NFV. In *OSDI*, 2016. USENIX.

[42] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud Scale Load Balancing. In *SIGCOMM*, 2013. ACM.

[43] S. Rajagopalan, D. Williams, and H. Jamjoom. Pico Replication: A High Availability Framework for Middleboxes. In *SoCC*, 2013. ACM.

[44] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. a. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker. Rollback-Recovery for Middleboxes. In *SIGCOMM*, 2015. ACM.

[45] R. P. Singh, T. Brecht, and S. Keshav. Towards VM Consolidation Using a Hierarchy of Idle States. In *VEE*, 2015. ACM.

[46] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. In *OSDI*, 2002. USENIX.

[47] A. Verma, J. Bagrodia, and V. Jaiswal. Virtual Machine Consolidation in the Wild. In *Middleware*, 2014. ACM.

[48] W. Voorsluys, J. Broberg, S. Venugopal, and R. Buyya. Cost of Virtual Machine Live Migration in Clouds: A Performance Evaluation. In *CloudCom*, 2009. Springer-Verlag.

[49] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif. Black-box and Gray-box Strategies for Virtual Machine Migration. In *NSDI*, 2007. USENIX.

[50] K. Ye, X. Jiang, D. Huang, J. Chen, and B. Wang. Live Migration of Multiple Virtual Machines with Resource Reservation in Cloud Computing Environments. In *CLOUD*, 2011. IEEE.

[51] X. Zhang, Z.-Y. Shae, S. Zheng, and H. Jamjoom. Virtual Machine Migration in an Over-committed Cloud. In *NOMS*, 2012. IEEE.

Cache Modeling and Optimization using Miniature Simulations

Carl A. Waldspurger
CachePhysics, Inc.
carl@cachephysics.com

Trausti Saemundson
CachePhysics, Inc.
trauzti@gmail.com

Irfan Ahmad
CachePhysics, Inc.
irfan@cachephysics.com

Nohhyun Park
Datos IO, Inc.
nohhyun.park@datos.io

Abstract

Recent approximation algorithms (*e.g.*, CounterStacks, SHARDS and AET) make lightweight, continuously-updated miss ratio curves (MRCs) practical for online modeling and control of LRU caches. For more complex cache-replacement policies, *scaled-down simulation*, introduced with SHARDS, offers a general method for emulating a given cache size by using a miniature cache processing a small spatially-hashed sample of requests.

We present the first detailed study evaluating the effectiveness of this approach for modeling non-LRU algorithms, including ARC, LIRS and OPT. Experiments with over a hundred real-world traces demonstrate that scaled-down MRCs are extremely accurate while requiring dramatically less space and time than full simulation.

We propose an efficient, generic framework for dynamic optimization using multiple scaled-down simulations to explore candidate cache configurations simultaneously. Experiments demonstrate significant improvements from automatic adaptation of parameters including the stack size limit in LIRS, and queue sizes in 2Q.

Finally, we introduce *SLIDE*, a new approach inspired by Talus that uses scaled-down MRCs to remove performance cliffs automatically. SLIDE performs shadow partitioning transparently within a single unified cache, avoiding the problem of migrating state between distinct caches when partition boundaries change. Experiments demonstrate that SLIDE improves miss ratios for many cache policies, with large gains in the presence of cliffs.

1 Introduction

Caches are ubiquitous in modern computing systems, improving system performance by exploiting locality to reduce access latency and offload work from contended storage systems and interconnects. However, caches are notoriously difficult to model. It is well-known that performance is non-linear in cache size, due to complex effects that vary enormously by workload. Techniques for accurate and efficient cache modeling are especially valuable to inform cache allocation and partitioning decisions, optimize cache parameters, and support goals including performance, isolation, and quality of service.

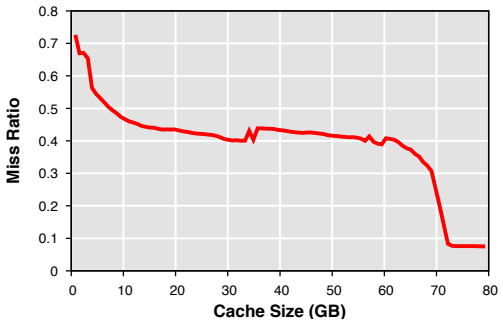


Figure 1: **Example MRC.** Miss-ratio curve for a production disk block trace using ARC cache algorithm. The ratio of cache misses to total references is plotted as a function of cache size.

1.1 Cache Modeling

Cache utility curves plot a performance metric as a function of cache size. Figure 1 shows an example miss-ratio curve (MRC), which plots the ratio of cache misses to total references for a workload (y-axis) as a function of cache size (x-axis). The miss ratio generally decreases as cache size increases, although complex algorithms such as ARC [14] and LIRS [9] can exhibit non-monotonic behavior due to imperfect dynamic adaptation.

MRCs are valuable for analyzing cache behavior. Assuming a workload exhibits reasonable stationarity at the time scale of interest, its MRC can also predict future performance. Thus, MRCs are powerful tools for optimizing cache allocations to improve performance and achieve service-level objectives [3, 11, 18, 22, 27, 28].

1.2 MRC Construction

Before the seminal paper by Mattson *et al.* [13], studies of memory and storage caching required running separate experiments for each cache size. Their key insight was that many replacement policies exhibit an inclusion property: given a cache C of size S, $C(S) \subseteq C(S+1)$. Such policies, which include LRU, LFU, and MRU, are referred to as *stack algorithms*. Mattson *et al.* introduced a method for such algorithms that constructs the entire MRC for all cache sizes in a single pass over a trace.

For a trace of length N containing M unique blocks, Mattson's algorithm takes $O(NM)$ time and $O(M)$ space. Efficient modern implementations of this algorithm have

an asymptotic cost of $O(N \log M)$ time and $O(M)$ space, employing a balanced tree to compute reuse distances and a hash table to accelerate lookups into this tree [16].

Recent advances [7, 20, 23, 26] have produced approximate methods that construct accurate MRCs with dramatically lower costs than exact methods. In particular, SHARDS [23] and AET [7] require only $O(N)$ time and $O(1)$ space, with a tiny footprint of approximately 1 MB.

Previously relegated to offline modeling, MRCs for stack algorithms can now be computed so inexpensively that they are practical for dynamic, online cache management, even in the most demanding environments. However, for more complex non-stack algorithms, such as ARC and LIRS, there are no known single-pass methods. As a result, separate runs are required for each cache size, similar to pre-Mattson modeling of LRU caches.

1.3 Cache Optimization

Low-cost online modeling of cache behavior using MRCs has many practical applications. Whereas a single cache instance runs with a single policy and a single set of configuration parameters, the ability to efficiently instantiate multiple concurrent models with different cache configurations offers a powerful generic framework for dynamic optimization. By simulating candidate cache configurations simultaneously, a system can quantify the impact of hypothetical parameter changes, so that the best settings can be applied to the actual cache.

This approach has the potential to overcome a key challenge in designing cache software today: policy and parameter tweaking is typically performed only at design time, considering a small number of benchmarks. Since no single configuration is best for all workloads, there is a significant optimization opportunity to instead adapt parameters automatically in live deployments.

A multi-model approach can help select the best general options, such as cache block size, write policy, or even replacement policy. The same method supports dynamic tuning of algorithm-specific parameters, such as queue sizes for 2Q [10] or LIRS [9].

Lightweight MRCs can be futher leveraged to guide efficient cache sizing, allocation, and partitioning for both individual workloads and complex multi-workload environments. For example, Talus shadow partitioning [3], which requires an MRC as input, can remove performance cliffs within a single workload, and improve cache partitioning across workloads.

1.4 Contributions

We make several key contributions over prior research in the areas of cache modeling and optimization:

Evaluate scaled-down simulation for complex policies To the best of our knowledge, scaled-down simulation is the *only* general approach capable of fast and accurate modeling of complex caching algorithms. We present the first detailed evaluation with non-LRU caching algorithms, including ARC, LIRS, and OPT. Our results indicate that sampling rates as low as 0.1% yield accurate MRCs with approximate miss ratio errors averaging much less than 0.01, at extremely low overhead.

New optimization framework We introduce a powerful new framework for optimizing cache performance dynamically by leveraging miniature cache simulations.

Transparent cliff removal We highlight challenges with Talus shadow partitioning for non-stack algorithms, and introduce SLIDE, a new approach that removes performance cliffs from such algorithms automatically and transparently – the first practical application of cliff removal techniques to complex cache algorithms.

New LIRS observations We describe previously-unreported parameter sensitivity and non-monotonic behavior with LIRS, and present a useful new optimization.

Although we focus on block-based storage systems, our techniques are broadly applicable to nearly any form of caching, including memory management in operating systems and hypervisors, application-level caches, key-value stores, and even hardware cache implementations.

The next section provides some background on non-stack caching algorithms. Section 3 describes our core scaled-down cache modeling technique, and presents a detailed evaluation of its accuracy and performance. Scaled-down caches are leveraged to optimize LIRS and 2Q by adapting algorithm parameters in Section 4. Section 5 introduces SLIDE, a new approach for removing performance cliffs, and demonstrates improvements with several cache policies. Related work is discussed in Section 6. Finally, we summarize our conclusions and highlight opportunities for future work in Section 7.

2 Non-stack Algorithms

Many modern caching algorithms outperform LRU on a wide range of workloads. Several, such as ARC, LIRS, and 2Q, treat blocks that have recently been seen only once differently from those that have been seen at least twice. Many policies employ *ghosts* – small metadata-only entries containing block identifiers, but not actual data. Some, like ARC, adapt to changes in workload patterns automatically. It is not surprising that such sophisticated policies are *non-stack algorithms* that violate the stack inclusion property. All caching algorithms aspire to close the gap with OPT, the unrealizable optimal policy.

2Q Inspired by LRU-K [17], Johnson and Shasha developed the 2Q algorithm [10]. As its name suggests, 2Q uses two queues: *A1* for blocks seen once and *Am* for blocks seen more than once. *A1* is split into *A1in*

and *A1out*, where *A1out* is a metadata-only ghost extension of *A1in*. 2Q promotes a block to *Am* only on a hit in *A1out*, so *A1in* behaves as a FIFO. The algorithm has two tunable parameters – the size of *A1in* relative to *Am*, and the size of *A1out* relative to the cache size.

ARC Megiddo and Modha introduced ARC, the adaptive replacement cache policy [14]. ARC is a self-tuning algorithm that manages both recently-used and frequently-used blocks in separate LRU lists: *T1* for blocks seen once, *T2* for blocks seen more than once, and their corresponding ghost extensions, *B1* and *B2*, which track metadata for recently-evicted blocks. Queue sizes change adaptively based on which gets more cache hits; there are no tunable parameters. ARC has been deployed widely in production systems, and is considered by many to be the "gold standard" for storage caching.

LIRS Jiang and Zhang developed LIRS, the low inter-reference recency set algorithm [9]. LIRS uses recency to estimate reuse distance when making replacement decisions. Blocks are categorized into high reuse-distance (HIR) and low reuse-distance (LIR) sets. All LIR blocks are resident but HIR blocks can be resident or ghosts. A block changes from HIR to LIR when its reuse distance is low compared to the current LIR set.

LIRS employs two LRU lists, called the S and Q stacks. Q contains all resident HIR blocks, and S contains LIR blocks as well as some resident HIR blocks and HIR ghosts. LIRS has two tunable parameters – the ratio of resident HIR and LIR blocks (the authors suggest 1% HIR), and the maximum size of S, which effectively bounds the number of ghosts. LIRS has been adopted by several production systems, including MySQL [25].

OPT Belady first described OPT, the theoretically optimal algorithm, also known as MIN [4, 1, 13]. OPT is a "clairvoyant" algorithm, since it relies on knowledge of future references to evict the block that will be reused the farthest in the future. Although OPT is actually a stack algorithm [21], it cannot be used to implement online eviction. Instead, OPT provides a bound on the performance of realizable algorithms.

3 Scaled-down Modeling

SHARDS [23] introduced single-pass techniques for constructing approximate MRCs based on randomized spatial sampling. References to representative locations are selected dynamically based on a function of their hash values. The "scaled down" reference stream is provided as input to a conventional single-pass MRC construction algorithm [13, 16] and the reuse distances it outputs are "scaled up" to adjust for the sampling rate.

While this approach works extremely well for stack algorithms such as LRU, there is no known single-pass method for non-stack caching algorithms. For such policies, a discretized MRC must be constructed by running separate simulations at many different cache sizes.

To support efficient modeling of *any* caching algorithm, the SHARDS authors proposed emulating a given cache size using a *miniature cache* running the full, unmodified algorithm over a small spatially-hashed sample of requests. Although a proof-of-concept experiment yielded promising results [23], there has been no detailed study of this approach. We present the first comprehensive evaluation of scaled-down simulation for modeling the sophisticated ARC, LIRS and OPT algorithms.

3.1 Miniature Simulations

A miniature simulation can emulate a cache with any specified size by scaling down both the actual cache size and its input reference stream. For example, consider modeling a cache with size S using a sampling rate R. A miniature simulation may emulate a cache of size S by scaling down the cache size to $R \cdot S$ and scaling down the reference stream using a hash-based spatial filter with sampling rate R. In practice, sampling rates on the order of $R = 0.01$ or $R = 0.001$ yield very accurate results, achieving huge reductions in space and time compared to a conventional full-size simulation.

More generally, scaled-down simulation need not use the same scaling factor for both the miniature cache size and its reference stream, although such configurations were not discussed when the technique was originally proposed [23]. The emulated cache size S_e, mini-cache size S_m, and input sampling rate R are related by $S_e = S_m/R$. Thus, S_e may be emulated by specifying a fixed rate R, and using a mini-cache with size $S_m = R \cdot S_e$, or by specifying a fixed mini-cache size S_m, and sampling its input with rate $R = S_m/S_e$. In practice, it is useful to enforce reasonable constraints on the minimum mini-cache size (*e.g.*, $S_m \geq 100$) and sampling rate (*e.g.*, $R \geq 0.001$) to ensure sufficient cache space and enough sampled references to simulate meaningful behavior.

3.1.1 Error Reduction

Like SHARDS, we apply a simple adjustment to reduce sampling error when computing the miss ratio for a miniature simulation. We have observed that when the number of sampled references, N_s, differs from the expected number, $E[N_s] = N \cdot R$, the sample set typically contains the wrong proportion of frequently-accessed blocks. To correct for this bias we divide the number of misses m by the *expected* number of references, instead of the actual number of references; *i.e.*, $m/E[N_s]$ is a better approximation of the true miss ratio than m/N_s.

3.1.2 Caches with Integrated Modeling

We have experimented with an alternative "unified" approach that integrates MRC construction into a live pro-

duction cache, without running separate simulations. Spatial hashing shards requests across a set of cache partitions, all serving actual requests. Several small partitions serve as *monitoring shards*, emulating multiple cache sizes within a small fraction of the overall cache.

An MRC can be generated on demand by simply accessing the miss ratios associated with each monitoring shard. Although integrated monitoring avoids additional simulation costs, we found that it typically degrades overall cache performance slightly, since most monitoring shards will not have efficient operating points.

3.2 Scaled-down MRCs

For non-stack algorithms, there are no known methods capable of constructing an entire MRC in a single pass over a trace. Instead, MRC construction requires a separate run for each point on the MRC, corresponding to multiple discrete cache sizes. Fortunately, we can leverage miniature caches to emulate each size efficiently.

We evaluate the accuracy and performance of our approach with three diverse non-LRU cache replacement policies: ARC, LIRS, and OPT. We developed efficient implementations of each in C, and validated their correctness against existing implementations [6, 8, 19].

We use a collection of 137 real-world storage block trace files, similar to those used in the SHARDS evaluation. These represent 120 week-long virtual disk traces from production VMware environments collected by CloudPhysics [23], 12 week-long enterprise server traces collected by Microsoft Research Cambridge [15], and 5 day-long server traces collected by FIU [12].

For our experiments, we use a 16 KB cache block size, and misses are read from storage in aligned, fixed-size 16 KB units. Reads and writes are treated identically, effectively modeling a simple write-back caching policy. We have also experimented with 4 KB blocks, modeling different write policies, and processing only read requests, all with similar results.

3.2.1 Accuracy

For each trace, we compute MRCs at 100 discrete cache sizes, spaced uniformly between zero and a maximum cache size. To ensure these points are meaningful, the maximum cache size is calculated as the aggregate size of all unique blocks referenced by the trace. This value was estimated during a separate, one-time pre-processing step for each trace, using fixed-size SHARDS [23].

To quantify accuracy, we compute the difference between the approximate and exact miss ratios at each discrete point on the MRC, and aggregate these into a mean absolute error (MAE) metric, as in related work [26, 23, 7]. The box plots[1] in Figure 2 show the MAE distributions for ARC, LIRS, and OPT with sampling rates $R = 0.01$ and $R = 0.001$. The average error is surprisingly small in all cases. For $R = 0.001$, the median MAE for each algorithm is below 0.005, with a maximum of 0.033. With $R = 0.01$, the median MAE for each algorithm is below 0.002, with a maximum of 0.012.

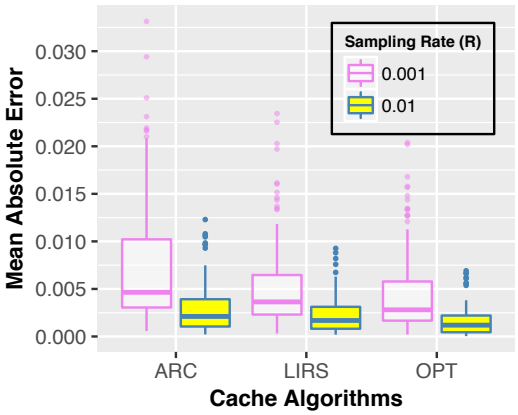


Figure 2: **Error Analysis.** Distribution of mean absolute error for all 137 traces with three algorithms (ARC, LIRS, OPT) at two different sampling rates ($R = 0.01$, $R = 0.001$).

Since the minimum cache size for LIRS is 200 blocks (to support the default 1% allocation to HIR), the LIRS MAE was calculated using this minimum size for some miss ratios, implying a higher sampling rate. Excluding these min-size runs, the median MAE for $R = 0.001$ is below 0.003, with a maximum of 0.025; for $R = 0.01$, the median is below 0.002, with a maximum of 0.009.[2]

Figure 3 contains 35 small plots that illustrate the accuracy of approximate MRCs with $R = 0.001$ on example traces with diverse MRC shapes and sizes. In most cases, the approximate and exact curves are nearly indistinguishable. The plots in Figure 4 show this accuracy with much greater detail for two example MSR traces. In all cases, miniature simulations model cache behavior accurately, including complex non-monotonic behavior by ARC and LIRS. These compelling results with such diverse algorithms and workloads suggest that scaled-down simulation is an extremely general technique capable of modeling nearly any caching algorithm.

3.2.2 Performance

For our performance evaluation, we used a platform configured with a six-core 3.3 GHz Intel Core i7-5820K processor and 32 GB RAM, running Ubuntu 14.04 (Linux kernel 4.4). Experiments compare traditional exact simulation with our lightweight scaled-down approach.

Resource consumption was measured using our five largest traces. We simulated three cache algorithms at

[1] The top and the bottom of each box represent the first and third quartiles. The thin whiskers show the min and max, excluding outliers.

[2] The reduced MAE may seem counter-intuitive. However, accuracy generally improves as the size of the scaled-down cache increases, and the excluded points were the smaller, less-accurate cache sizes.

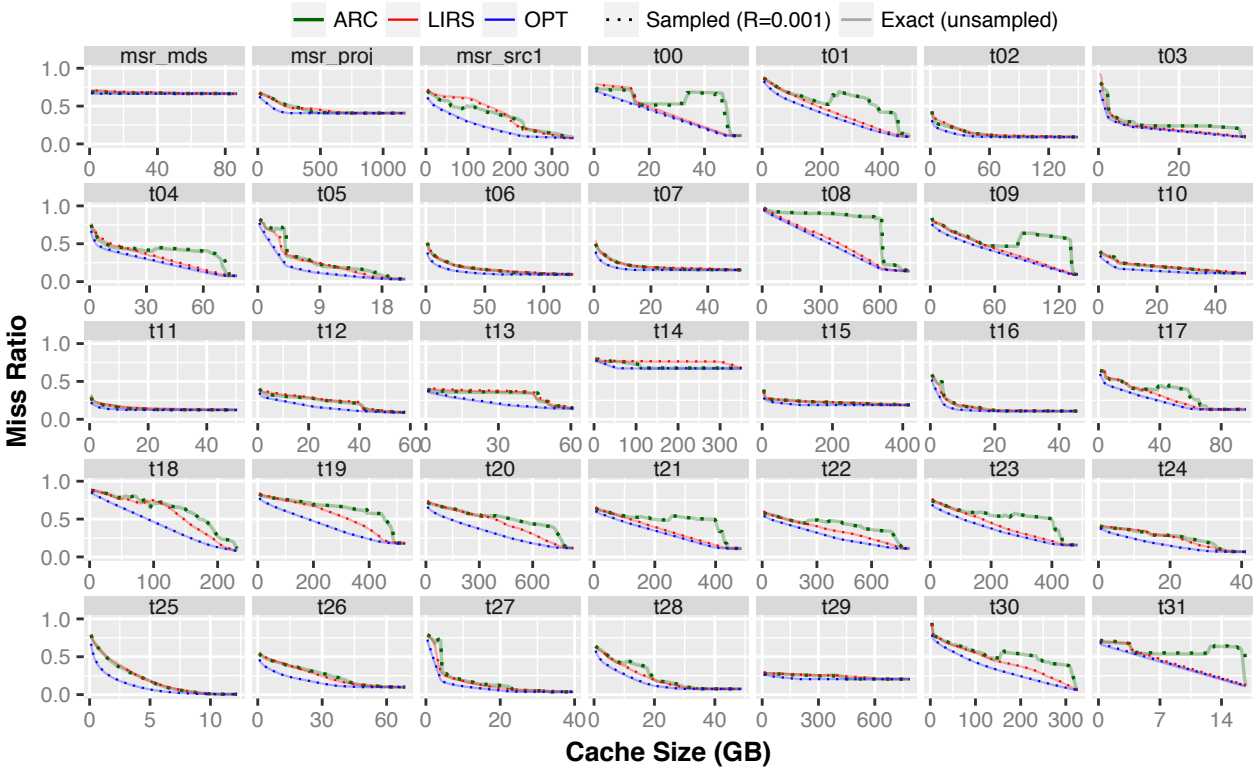


Figure 3: **Diverse MRCs: Exact vs. Miniature.** Exact and approximate MRCs for 35 representative traces: three named MSR workloads [15], and the CloudPhysics workloads labeled `t00`–`t31` in the SHARDS evaluation [23]. Approximate MRCs are constructed using scaled-down simulation with sampling rate $R = 0.001$. Each color represents a different cache algorithm.

five emulated sizes S_e (8 GB, 16 GB, 32 GB, 64 GB and 128 GB), using multiple sampling rates R (1, 0.1, 0.01 and 0.001) for a total of 60 experiments per trace. We repeated each run five times, and report average values.

Unsurprisingly, the memory footprint[3] for cache simulation is a simple linear function consisting of fixed overhead (for policy code, libraries, etc.) plus variable space. For ARC and LIRS, the variable component is proportional to the cache size, $R \cdot S_e$. For OPT, which must track all future references, it is proportional to the number of sampled references, $R \cdot N$. Table 1 reports the fixed and variable components of the memory overhead determined by linear regression ($r^2 > 0.99$). As expected, accurate results with $R = 0.001$ require 1000× less space than full simulation, excluding the fixed overhead.

We also measured the CPU usage[4] consumed by our single-threaded cache implementations with both exact and scaled-down simulations for ARC, LIRS and OPT. As shown in Figure 5, runtime consists of two main components: cache simulation time, which is roughly linear in R, and the sampling overhead involving hashing and trace file I/O, which is roughly constant; MurmurHash [2] is invoked for each reference to determine if it should be sampled. The total runtime from each experiment was decomposed by running a corresponding experiment with a pre-sampled trace file, removing the overhead of hashing and most I/O costs.

	Linear Function		Example Trace (t22)	
Policy	**Fixed**	**Variable**	**R=0.001**	**R=1**
ARC	1.37 MB	71 B	1.57 MB	284 MB
LIRS	1.59 MB	75 B	1.80 MB	301 MB
OPT	7.10 MB	37 B	19.55 MB	18,519 MB

Table 1: **Memory Footprint.** Memory usage for ARC and LIRS is linear in the cache size, $R \cdot S_e$, while for OPT, it is linear in the number of sampled references, $R \cdot N$. Measured values are shown for CloudPhysics trace `t22` with S_e = 64 GB.

Overall, scaled-down simulation with $R = 0.001$ requires about 10× less CPU time than full simulation, and achieves throughput exceeding 53 million references per second for ARC and LIRS, and 39 million references per second for OPT. Fortunately, for multi-model optimization, hash-based sampling costs are incurred only once, not for each mini-cache. In an actual production cache, the cost of data copying would dwarf the hashing overhead, which represents a larger fraction of the fast cache simulation operations that manipulate only metadata.

[3]The peak resident set size was obtained from the Linux procfs node `/proc/<pid>/status` immediately before terminating.

[4]CPU time was obtained by adding the user and system time components reported by `/usr/bin/time`.

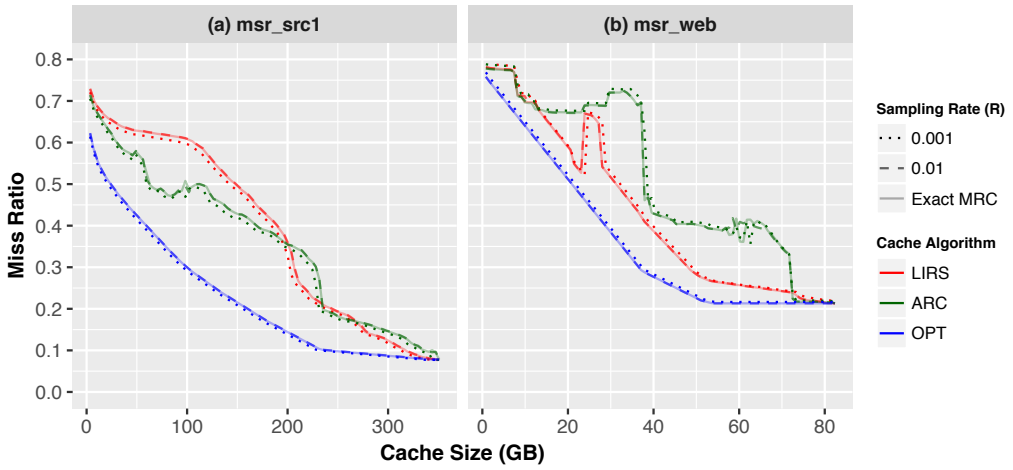


Figure 4: **Detailed MRCs.** Approximate MRCs for two enterprise storage traces [15] with three different algorithms. Miniature simulations capture cache behavior accurately, including complex non-monotonicity. All MAEs for $R = 0.001$ are less than 0.011.

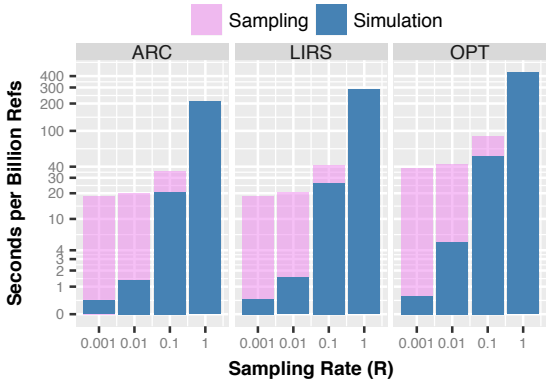


Figure 5: **Runtime.** Time to process one billion references as a function of R. *Sampling* represents time spent selecting references to simulate; *Simulation* is time spent actually executing the cache policy.

4 Adapting Cache Parameters

Our generic multi-model optimization framework leverages miniature simulations to adapt cache parameters dynamically. The impact of multiple candidate parameter values is evaluated periodically, and the best setting is applied to the actual cache. We present example optimizations that adapt cache parameters automatically for two well-known replacement algorithms: LIRS and 2Q.

While MRCs are typically stable over short time periods, they frequently vary over longer intervals. To adapt dynamically to changing workload behavior, we divide the input reference stream into a series of *epochs*. Our experiments use epochs consisting of one million references, although many alternative definitions based on wall-clock time, evictions, or other metrics are possible.

After each epoch, we calculate an exponentially-weighted moving average (EWMA) of the miss ratio for each mini-cache, to balance historical and current cache behavior. Our experiments use an EWMA weight of 0.2 for the current epoch. The parameter value associated with the mini-cache exhibiting the lowest smoothed miss ratio is applied to the actual cache for the next epoch.

4.1 LIRS Adaptation

As discussed in Section 2, the LIRS S stack is LRU-ordered and contains a mix of LIR blocks, resident HIR blocks and non-resident HIR blocks (ghosts). This queue tracks the internal stack-distance ordering between HIR and LIR blocks. A HIR block is reclassified as LIR if it is referenced when it has a stack distance lower than that of the oldest LIR block. During this "status switch", the oldest LIR block is changed to HIR, evicted from S, and inserted into Q. After the status switch, a pruning operation removes all HIR blocks from the tail of S.

The default LIRS algorithm allows S to grow without bound on a sequence of misses. To address this issue, the authors suggest limiting the size of S; to enforce that limit, the oldest HIR ghost is evicted from S once the size exceeds the limit. We denote[5] this bound by f, relative to the cache size c; the total size of S is limited to $c \cdot f$. The LIRS paper experimented with a few values of f and reported that even low values such as $f = 2$ work well. Our evaluation of scaled-down modeling accuracy in Section 3.2 uses $f = 2$, so that LIRS tracks roughly the same number of ghost entries as ARC.

Code Optimization We started our work on LIRS with a C implementation obtained from the authors [8]. However, this code enforced the S size limit by always searching for the oldest HIR ghost starting from the tail of S. Since this caused excessive overhead with our large traces, we developed a simple optimization that stores

[5]This limit was not explicitly named in the LIRS paper.

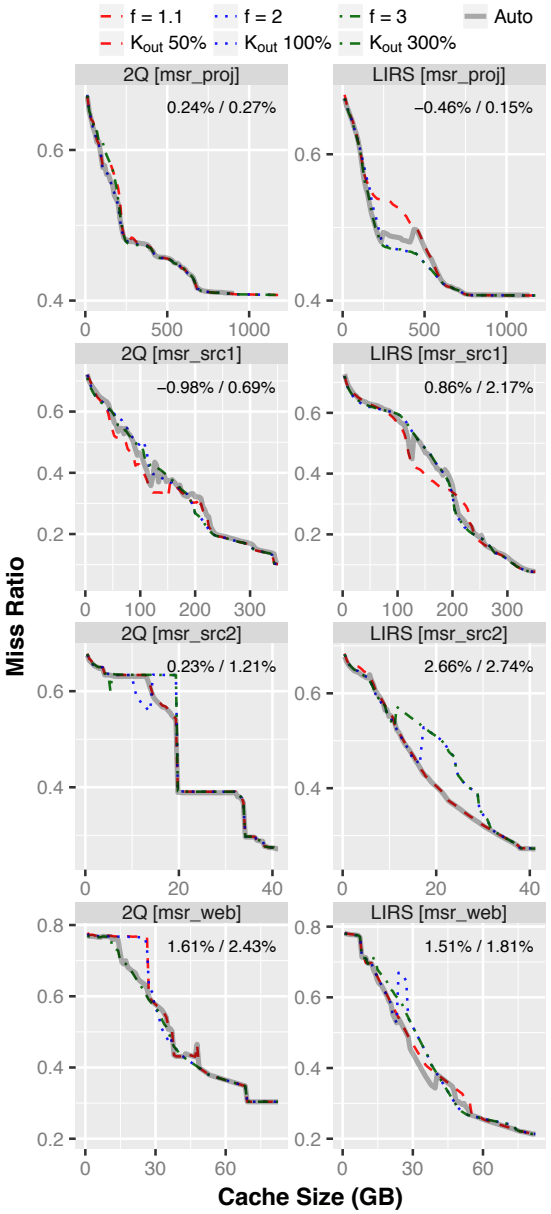


Figure 6: **Adaptive Parameter Tuning.** Dynamic multi-model optimization results for four example traces. Adaptation selects good values for 2Q (*Kout*) and LIRS (f) at most cache sizes with potential gains. The percentages (upper right) show the actual adaptation gain (*vs.* f=2, 50% *Kout*) and the potential gain (best per-cache-size values), averaged over all cache sizes.

a pointer to the entry previous to the last-removed HIR ghost.[6] It is guaranteed that no HIR ghost can appear after this element because entries are always added to the head of S. If the entry associated with this pointer is removed from the middle of S, such as on a hit, it is simply updated to the previous entry in the queue. We describe a similar optimization for SLIDE evictions in Section 5.5.

[6]This optimization met with approval from the LIRS authors [8].

Non-monotonicity Although the authors reported that LIRS does not suffer from non-monotonic behavior [9], we have observed it with several of our workloads when limiting the size of S. For example, Figure 4 reveals a prominent region for the `msr_web` trace where increasing the LIRS cache size results in a higher miss ratio. Interestingly, the degree of non-monotonicity varies with f, and there appear to be workload-dependent values that eliminate this behavior. For example, Figure 6 shows that `msr_src1`, `msr_src2` and `msr_web` perform well with $f = 1.1$, while $f = 3.0$ is best for `msr_proj`.

Automatic Tuning We use our multi-model optimization approach to adapt the LIRS f value dynamically for a subset of the traces described in Section 3.2. For each workload, five scaled-down simulations are performed with different values for f: 1.1, 1.5, 2.0, 2.5 and 3.0. Each simulation emulates the same cache size, equal to the size of the actual cache, with a fixed sampling rate $R = 0.005$. After each epoch consisting of 1M references, the miss ratios for each mini-cache are examined, and the best f value is applied to the actual cache.

Experiments The goal of automatic LIRS adaptation is to find the best value of f for each cache size. These ideal[7] static settings form an MRC that traces the lower envelope of the curves for different static f values. Actual and potential gains are computed as the mean signed difference across all cache sizes relative to the curve with the default fixed value $f = 2$. Potential gain is based on the best static f value for each cache size.

Figure 6 presents results for four representative MSR traces. Among all 12 MSR traces, `msr_src2` shows the best actual and potential gains; the worst case for adaptation is the net loss for `msr_proj`. For `msr_web` and `msr_src2`, adaptation converges on the best $f = 1.1$, and realizes 83–97% of the potential gain. The average actual *vs.* potential improvement across the MSR traces is 0.37% / 0.60%; adaptation captures the majority of possible gains. Results are mixed for traces like `msr_src1`, with some regressions, despite an overall gain. We are experimenting with automatic disabling of ineffective adaptation; early results show a small gain for `msr_proj`.

LIRS Observations For workloads with a single large working-set knee (*e.g.*, trace `t08` in Figure 3), the LIRS and OPT MRCs are often surprisingly close. LIRS appears to trace the convex hull of the LRU curve, slightly above OPT. This behavior is not intuitive, since LIRS has no knowledge of the knee, where the miss ratio drops suddenly once the working set fits. The explanation is that some blocks initially get stuck in the LIR set, and no later blocks have a low enough reuse distance to replace

[7]Although adaptation tends to converge on a single f value, selecting the best value for each individual epoch may yield a lower *dynamic* optimum. However, the combinatorics make this infeasible to simulate.

them. During another pass over the working set, accesses to these blocks will be hits. Thus, LIRS can effectively remove some cliffs by trapping blocks in the LIR set.

4.2 2Q Adaptation

The 2Q algorithm is not adaptive, so its queue sizes are specified manually. The authors suggest allocating 25% of the cache space to *A1in* and 75% to *Am*. They also suggest sizing the number of ghosts in *A1out* to be 50% of the elements in the cache. The 2Q paper defines the parameter *Kin* as the size of *A1in*, and *Kout* as the size of *A1out*, the ghost queue for blocks seen only once.

Comparing 2Q and LIRS, *Am* is similar to the subset of the LIRS *S* stack containing LIR blocks, *A1in* is comparable to the LIRS *Q* stack, and *A1out* is similar to the subset of the LIRS *S* stack containing HIR ghost blocks. While LIRS performs well allocating just 1% of its space to *Q*, 2Q needs a higher percentage for *A1in*. The sizing of *A1out* in 2Q is similar to f-adaptation in LIRS.

Since 2Q does not adapt its queue sizes dynamically, we again employ multi-model optimization, using eight scaled-down simulations with $R = 0.005$, 25% *Kin*, and *Kout* parameters between 50% and 300%. After each epoch consisting of 1M references, the best *Kout* value is applied to the actual cache. Automatic adaptation is designed to find the optimal *Kout* for each cache size. As in Section 4.1, we compute gain relative to the area between the default curve with fixed 50% *Kout* and the lower envelope of all the curves with static *Kout* values.

Figure 6 shows adaptation works well for `msr_web`, which has the best actual and potential gains over all 12 MSR traces; the auto-adapted curve tracks the lower envelope closely, capturing 66% of the possible static gain. For traces like `msr_proj` that are not very sensitive to *Kout*, adaptation shows modest absolute gains. The worst case is the significant loss for `msr_src1`, although the auto-disabling extension mentioned earlier results in a small gain. Averaged over all 12 MSR traces, the actual *vs.* potential improvement is 0.10% / 0.45%.

5 SLIDE

SLIDE is a new approach inspired by Talus [3] that leverages scaled-down MRCs to remove performance cliffs. We describe challenging issues with applying Talus to non-LRU policies, and explain how SLIDE resolves them. We then present efficient SLIDE implementation techniques that support transparent shadow partitioning within a single unified cache.

5.1 Talus Inspiration

Talus [3] is a technique that removes cache performance cliffs using hash-based partitioning. It divides the reference stream for a single workload into two shadow partitions, *alpha* and *beta*, steering a fraction ρ of references to the alpha partition. Each partition can be made to emulate the performance of a smaller or larger cache by controlling its size and input load.

Talus requires the workload's MRC as an input. The partition sizes N_α and N_β, and their respective loads, ρ and $1-\rho$, are computed in a clever manner that ensures their combined aggregate miss ratio lies on the convex hull of the MRC. Although Talus was introduced in the context of hardware processor caches, a similar idea has also been applied to key-value web caches [5].

We view the hash-based partitioning employed by Talus for removing performance cliffs and the hash-based monitoring introduced with SHARDS for efficient MRC construction as two sides of the same coin. Both rely on the property that hash-based sampling produces a smaller reference stream that is statistically self-similar to the original stream. The ability to construct MRCs using hash-based sampling was not recognized by the Talus authors, who emphasized that no known methods could generate MRCs inexpensively for non-stack algorithms.

5.2 Challenges with Non-LRU MRCs

As noted by the Talus authors, a key challenge with applying Talus to non-stack algorithms is the need to construct MRCs efficiently in an online manner. This problem is solved by using the scaled-down modeling techniques described in Section 3. As with parameter adaptation described in Section 4, we divide the input reference stream into a series of epochs. After each epoch, we construct a discretized MRC from multiple scaled-down simulations with different cache sizes, smoothing each miss ratio using an EWMA. We then identify the subset of these miss ratios that form the convex hull for the MRC, and compute the optimal partition sizes and loads using the same inexpensive method as Talus.

In theory, the combination of scaled-down MRC construction and Talus shadow partitioning promises to improve the performance of *any* caching policy by interpolating efficient operating points on the convex hulls of workload MRCs. In practice, we encountered several additional challenges while trying to implement Talus for caching algorithms such as ARC and LIRS.

5.3 Challenges with Non-LRU Partitioning

Talus requires distinct cache instances for its separate alpha and beta partitions, which together have a fixed total size. This hard division becomes problematic in a system where the partition boundaries change dynamically in response to an MRC that evolves over time. Similarly, when ρ changes dynamically, some cache entries may reside in the "wrong" partition based on their hash values.

These issues were not discussed in the Talus paper.

We initially tested simple strategies to address these issues. For example, removing cache entries eagerly when decreasing the size of a partition, and reallocating the reclaimed space to the other partition. Or migrating entries across partitions eagerly to ensure that each resides in the correct partition associated with its hash. Such eager strategies performed poorly, as migration checks and actions are expensive, and data may be evicted from one partition before the other needs the space. Moreover, it's not clear how migrated state should be integrated into its new partition, even for a simple policy like LRU, since list positions are not ordered across partitions.

A lazy strategy for reallocation and migration generally fares better. Cache entries can be reclaimed from an over-quota partition on demand, and entries residing in incorrect partitions migrated only on hits. However, this approach adds non-trivial complexity to the core caching logic. More importantly, while migrating to the MRU position on a hit seems reasonable for an LRU policy, it's not clear how to merge state appropriately for more general algorithms. Some policies do not even specify how to transform state to support dynamic resizing.

5.4 Transparent Shadow Partitioning

We developed a new approach called *SLIDE* (*S*harded *L*ist with *I*nternal *D*ifferential *E*viction) to address these challenges. In contrast to Talus, SLIDE maintains a single unified cache, and defers partitioning decisions until eviction time. SLIDE conveniently avoids the resizing, migration, and complexity issues discussed above.

A *SLIDE list* is a new abstraction that serves as a drop-in replacement for the standard LRU list used as a common building block by many sophisticated algorithms, including ARC, LIRS, and 2Q. Since SLIDE interposes on primitive LRU operations that add (insert-at-head), reference (move-to-head), and evict (remove-from-tail) entries, it is completely transparent to cache-replacement decisions. An unmodified cache algorithm can support Talus-like partitioning by simply relinking to substitute SLIDE lists for LRU[8] lists. We have successfully optimized ARC (*T1*, *T2*, *B1* and *B2*), LIRS (*S* and *Q*), 2Q (*Am*, *A1in* and *A1out*), and LRU in this manner.

5.5 SLIDE Lists

A SLIDE list is implemented by extending a conventional doubly-linked LRU list. All list entries remain ordered from MRU (head) to LRU (tail). Each entry is augmented with a compact hash[9] of its associated location identifier (*e.g.*, block number or memory address).

[8]SLIDE also works with FIFO lists, such as the 2Q *A1in* queue; a referenced entry simply isn't moved to the head of the list.

[9]Our default implementation uses small 8-bit hashes, providing better than 0.4% resolution for performing hash-based partitioning.

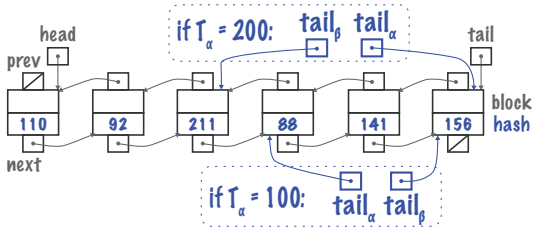


Figure 7: **SLIDE List.** Extensions (highlighted) to a doubly-linked list include hash values plus $tail_\alpha$ and $tail_\beta$ pointers used to find the LRU blocks in the alpha and beta partitions. Different dynamic thresholds T_α illustrate flexible partitioning.

This hash value is compared to the current threshold T_α to classify the entry as belonging to either the alpha or beta "partition". This makes the SLIDE partition boundary more flexible than the hard partitions in Talus.

As depicted in Figure 7, in addition to the usual head and tail pointers, SLIDE maintains two new tail pointers, $tail_\alpha$ and $tail_\beta$. To evict from alpha, the LRU alpha entry is located by walking the list backwards from $tail_\alpha$ until an entry with a hash value below T_α is found. Similarly, an eviction from beta starts with $tail_\beta$ and continues until an entry with a hash value at or above T_α is found.

The tail-search pointers $tail_\alpha$ and $tail_\beta$ are initialized to `NULL`, which indicates that searches should begin from the absolute tail of the combined list. They are updated lazily during evictions, and to skip over entries that are moved to the MRU position on a hit. Since entries are added only to the head of the LRU-ordered list, the amortized cost for these updates is $O(1)$, as each tail pointer traverses a given entry only once.

Many LRU implementations maintain a count of the number of entries in the list. A SLIDE list also tracks N_α, the number of entries that currently belong to the alpha partition. The SLIDE configuration operation specifies both ρ and a target size for alpha, expressed as a fraction F_α of the total number of entries, N_{tot}. During an eviction, an entry is preferentially removed from the alpha partition if it is over quota (*i.e.*, $N_\alpha > F_\alpha \cdot N_{tot}$), or otherwise from the beta partition. If the preferred victim partition is empty, then the absolute LRU entry is evicted.

It is not obvious that substituting SLIDE lists for the internal lists within non-stack algorithms will approximate hard Talus partitions. The basic intuition is that configuring each SLIDE list with identical values of F_α and ρ will effectively divide the occupancy of each individual list – and therefore divide the entire aggregate algorithm state – to achieve the desired split between alpha and beta. As with Talus, this depends on the statistical self-similarity property of hash-based spatial sampling. While SLIDE may differ from strict Talus partitioning, it empirically works well for ARC, LIRS, 2Q, and LRU.

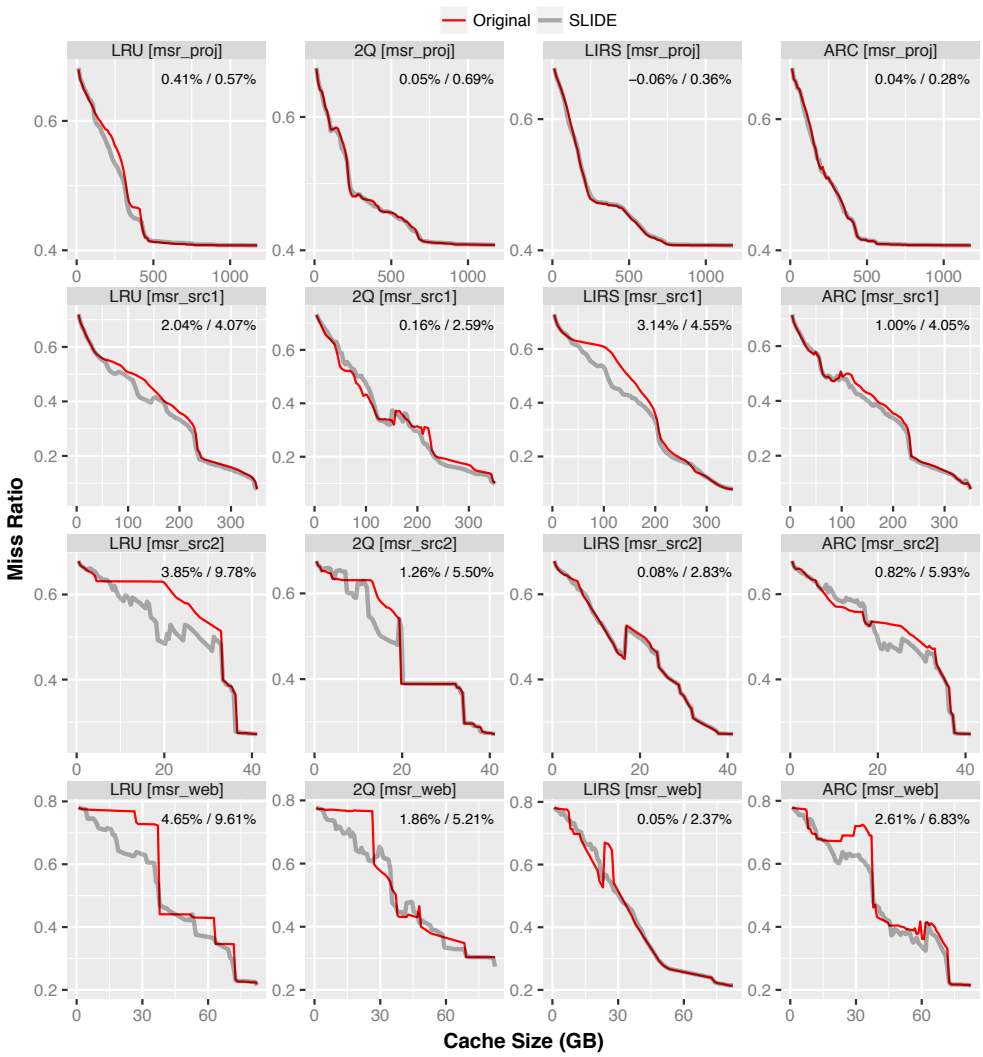


Figure 8: **SLIDE Cliff Removal.** Results for four traces using scaled-down MRCs from seven mini-cache simulations. SLIDE improves the miss ratio for LRU, 2Q, LIRS and ARC caches at most sizes with potential gains, but does exhibit some regressions. The percentages (upper right) show the actual SLIDE gain and the potential gain (ideal convex hull) averaged over all cache sizes.

5.6 SLIDE Reconfiguration

Periodic reconfiguration may move partition boundaries dynamically, changing the threshold T_α. To support constant-time recomputation of N_α, SLIDE optionally maintains an array of counts tracking the number of entries associated with each hash value.[10]

A change to T_α must also reset the $tail_\alpha$ and $tail_\beta$ search pointers, as later entries may have been reclassified to different partitions. Although not guaranteed to be $O(1)$, one pointer must be the same as the global tail, and the expected number of entries the other must re-traverse is $1/F_\alpha$, assuming a uniform hash distribution. This will typically be small compared to the epoch length, even for heavily-skewed partitions; F_α could also be bounded.

[10] An array of 256 counts for our implementation with 8-bit hashes.

5.7 Experiments

We evaluate the effectiveness of SLIDE using a subset of the traces described in Section 3.2. For each workload, a separate experiment is performed at 100 cache sizes. For each size, a discrete MRC is constructed via multiple scaled-down simulations with sampling rate $R = 0.005$. SLIDE is reconfigured after each 1M-reference epoch, using 0.2 as the EWMA weight for the current epoch.

Seven emulated cache sizes are positioned exponentially around the actual size, using relative scaling factors of 1/8, 1/4, 1/2, 1, 2, 4, and 8. For $R = 0.005$, the mini-cache metadata is approximately 8% of the actual metadata size (R times the sum of the scaling factors). For a 16 KB cache block size and 64 B metadata entries, this represents less than 0.04% of total memory consumption.

Many alternative configurations can provide differ-

ent time-space tradeoffs, *e.g.*, fixed-size variable-*R* mini-caches, as described in Section 3.1. Similarly, increasing the number of emulated cache sizes generally yields more accurate MRCs and improves SLIDE results, at the cost of additional mini-cache resource consumption.

Figure 8 plots the results of SLIDE performance cliff removal for four representative MSR traces with LRU[11], 2Q, LIRS and ARC policies. Ideally, SLIDE would trace the convex hull of the original MRC. In practice, this is not attainable, since the MRC evolves dynamically, and its few discrete points yield a crude convex hull. For each plot, we show both the actual SLIDE gain and the potential gain on the convex hull, each computed as the mean signed difference across all cache sizes from the original curve. We also characterize the larger set of all 12 MSR traces, although this metric often averages out more significant differences visible in the plots.

As expected, gains are largest for workloads with non-trivial cliffs, such as `msr_src2` and `msr_web`; SLIDE reduces their miss ratios by more than 10% in many cases. For the larger set of MSR traces, the best-case actual *vs.* potential gains are 4.65% / 9.78% (LRU), 1.86% / 5.50% (2Q), 3.14% / 4.55% (LIRS) and 2.61% / 6.84% (ARC). The average actual *vs.* potential improvements are 0.88% / 2.09% (LRU), 0.26% / 1.30% (2Q), 0.23% / 0.89% (LIRS) and 0.36% / 1.47% (ARC). Overall, SLIDE captures a reasonable fraction of possible gains.

For traces such as `msr_proj`, where the original MRC is nearly convex, SLIDE provides little improvement. For a few traces like `msr_src1`, results are mixed, with SLIDE improving many policies and cache sizes, but degrading others slightly. Across all 12 MSR traces and all four policies, the worst-case gain is -0.14%. As future work, we are extending SLIDE to disable itself dynamically to prevent losses and yield Pareto improvements.

6 Related Work

Research on cache modeling and MRC construction has focused on LRU and stack algorithms [13, 16, 26, 23, 7]. Modeling non-stack algorithms requires offline cache simulations with extremely high resource consumption, making online analysis and control impractical.

As explained in Section 3, basic scaled-down simulation was first introduced with our prior work on SHARDS [23], but there has been no detailed study of this approach. To the best of our knowledge, scaled-down simulation with miniature caches is the *only* approach that can model complex algorithms efficiently.

Our automatic adaptation is motivated by the observation that no single set of cache parameters performs well across all workloads. SOPA [24] is a cache framework for *inter-policy* adaptation. It collects a full trace during an evaluation period, replays it into simulators for multiple candidate policies, and adopts the best one. To facilitate policy switches, SOPA maintains a separate LRU-ordered list of all cached blocks. Blocks are replayed in LRU-to-MRU order, helping the new algorithm reconstruct recency metadata, but any frequency or ghost state is lost in translation. Our techniques are complementary, and could reduce SOPA analysis overhead significantly.

SLIDE, inspired by Talus [3], uses MRCs to remove cache performance cliffs. Section 5 presents a detailed comparison, and explains how SLIDE overcomes the challenges of applying Talus to non-LRU policies.

Cliffhanger [5] removes performance cliffs from web memory caches without an explicit miss-ratio curve. Assuming a full MRC is too expensive, limited-size shadow queues instead estimate its gradient, with the sign of the second derivative identifying a cliff. A significant limitation is that Cliffhanger can only scale a single cliff, which must be located within the limited visibility of its shadow queues. Although the authors state that Cliffhanger could work with any eviction policy, their algorithms and experiments are specific to LRU. It is not clear how to apply their shadow-queue technique to more complex caching policies, especially given the challenges identified in Section 5.3. Non-monotonicity may also present problems; even a small local bump in the MRC could be misinterpreted as the single cliff to be removed.

7 Conclusions

We have explored using miniature caches for modeling and optimizing cache performance. Compelling experimental results demonstrate that scaled-down simulation works extremely well for a diverse collection of complex caching algorithms – including ARC, LIRS, 2Q and OPT – across a wide range of real-world traces. This suggests our technique is a robust method capable of modeling nearly any cache policy accurately and efficiently.

Lightweight modeling of non-stack algorithms has many practical applications, including online analysis and control. We presented a general method that runs multiple scaled-down simulations to evaluate hypothetical configurations, and applied it to optimize LIRS and 2Q parameters automatically. We also introduced SLIDE, a new technique that performs Talus-like performance cliff removal transparently for complex policies.

Miniature caches offer the tantalizing possibility of improving performance for most caching algorithms on most workloads automatically. We hope to make additional progress in this direction by exploring opportunities to refine and extend our optimization techniques.

[11] SLIDE is very effective for LRU, and could use SHARDS or AET to construct MRCs more efficiently for a pure LRU policy.

Acknowledgments Thanks to CloudPhysics for helping to make this work possible, and to John Blumenthal, Jeff Hausman, Jim Kleckner, Xiaojun Liu, and Richard Sexton for their encouragement and support. Thanks also to the anonymous reviewers and our shepherd Timothy Wood for their valuable feedback and suggestions.

References

[1] AHO, A. V., DENNING, P. J., AND ULLMAN, J. D. Principles of Optimal Page Replacement. *J. ACM 18*, 1 (Jan. 1971), 80–93.

[2] APPLEBY, A. SMHasher and MurmurHash. `https://code.google.com/p/smhasher/`.

[3] BECKMANN, N., AND SANCHEZ, D. Talus: A Simple Way to Remove Cliffs in Cache Performance. In *Proceedings of the 21st international symposium on High Performance Computer Architecture (HPCA-21)* (February 2015).

[4] BELADY, L. A. A Study of Replacement Algorithms for Virtual Storage Computers. *IBM Systems Journal 5*, 2 (1966), 78–101.

[5] CIDON, A., EISENMAN, A., ALIZADEH, M., AND KATTI, S. Cliffhanger: Scaling Performance Cliffs in Web Memory Caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016), USENIX Association, pp. 379–392.

[6] GRYSKI, D. go-arc git repository. `https://github.com/dgryski/go-arc/`.

[7] HU, X., WANG, X., ZHOU, L., LUO, Y., DING, C., AND WANG, Z. Kinetic Modeling of Data Eviction in Cache. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference* (Berkeley, CA, USA, 2016), USENIX ATC '16, USENIX Association, pp. 351–364.

[8] JIANG, S. LIRS source code. Private communication, Oct 2016.

[9] JIANG, S., AND ZHANG, X. LIRS: An Efficient Low Interreference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2002), SIGMETRICS '02, ACM, pp. 31–42.

[10] JOHNSON, T., AND SHASHA, D. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1994), VLDB '94, Morgan Kaufmann Publishers Inc., pp. 439–450.

[11] KOLLER, R., MASHTIZADEH, A. J., AND RANGASWAMI, R. Centaur: Host-Side SSD Caching for Storage Performance Control. *2015 IEEE International Conference on Autonomic Computing (ICAC)* (2015), 51–60.

[12] KOLLER, R., AND RANGASWAMI, R. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. *Trans. Storage 6*, 3 (Sept. 2010), 13:1–13:26.

[13] MATTSON, R. L., GECSEI, J., SLUTZ, D. R., AND TRAIGER, I. L. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal 9*, 2 (June 1970), 78–117.

[14] MEGIDDO, N., AND MODHA, D. S. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2003), FAST '03, USENIX Association, pp. 115–130.

[15] NARAYANAN, D., DONNELLY, A., AND ROWSTRON, A. Write Off-loading: Practical Power Management for Enterprise Storage. *Trans. Storage 4*, 3 (Nov. 2008), 10:1–10:23.

[16] NIU, Q., DINAN, J., LU, Q., AND SADAYAPPAN, P. PARDA: A Fast Parallel Reuse Distance Analysis Algorithm. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium* (Washington, DC, USA, 2012), IPDPS '12, IEEE Computer Society, pp. 1284–1294.

[17] O'NEIL, E. J., O'NEIL, P. E., AND WEIKUM, G. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1993), SIGMOD '93, ACM, pp. 297–306.

[18] QURESHI, M. K., AND PATT, Y. N. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2006), MICRO 39, IEEE Computer Society, pp. 423–432.

[19] SAEMUNDSSON, T. cache algorithm git repository. `https://github.com/trauzti/cache/`.

[20] SAEMUNDSSON, T., BJORNSSON, H., CHOCKLER, G., AND VIGFUSSON, Y. Dynamic Performance Profiling of Cloud Caches. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2014), SOCC '14, ACM, pp. 28:1–28:14.

[21] SALTZER, J. H., AND KAASHOEK, M. F. *Principles of Computer System Design: An Introduction*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2009.

[22] STEFANOVICI, I., THERESKA, E., O'SHEA, G., SCHROEDER, B., BALLANI, H., KARAGIANNIS, T., ROWSTRON, A., AND TALPEY, T. Software-defined Caching: Managing Caches in Multi-tenant Data Centers. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (New York, NY, USA, 2015), SoCC '15, ACM, pp. 174–181.

[23] WALDSPURGER, C. A., PARK, N., GARTHWAITE, A., AND AHMAD, I. Efficient MRC Construction with SHARDS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2015), FAST '15, USENIX Association, pp. 95–110.

[24] WANG, Y., SHU, J., ZHANG, G., XUE, W., AND ZHENG, W. SOPA: Selecting the Optimal Caching Policy Adaptively. *Trans. Storage 6*, 2 (July 2010), 7:1–7:18.

[25] WIKIPEDIA. LIRS caching algorithm — Wikipedia, the free encyclopedia, 2017. [Online; accessed 22-Jan-2017].

[26] WIRES, J., INGRAM, S., DRUDI, Z., HARVEY, N. J. A., AND WARFIELD, A. Characterizing Storage Workloads with Counter Stacks. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI '14, USENIX Association, pp. 335–349.

[27] ZHAO, W., AND WANG, Z. Dynamic Memory Balancing for Virtual Machines. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2009), VEE '09, ACM, pp. 21–30.

[28] ZHOU, P., PANDEY, V., SUNDARESAN, J., RAGHURAMAN, A., ZHOU, Y., AND KUMAR, S. Dynamic Tracking of Page Miss Ratio Curve for Memory Management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2004), ASPLOS XI, ACM, pp. 177–188.

Hyperbolic Caching: Flexible Caching for Web Applications

Aaron Blankstein*, Siddhartha Sen†, and Michael J. Freedman*
* Princeton University, † Microsoft Research

Abstract

Today's web applications rely heavily on caching to reduce latency and backend load, using services like Redis or Memcached that employ inflexible caching algorithms. But the needs of each application vary, and significant performance gains can be achieved with a tailored strategy, e.g., incorporating cost of fetching, expiration time, and so forth. Existing strategies are fundamentally limited, however, because they rely on data structures to maintain a total ordering of the cached items.

Inspired by Redis's use of random sampling for eviction (in lieu of a data structure) and recent theoretical justification for this approach, we design a new caching algorithm for web applications called *hyperbolic caching*. Unlike prior schemes, hyperbolic caching decays item priorities at variable rates and continuously reorders many items at once. By combining random sampling with lazy evaluation of the hyperbolic priority function, we gain complete flexibility in customizing the function. For example, we describe extensions that incorporate item cost, expiration time, and windowing. We also introduce the notion of a *cost class* in order to measure the costs and manipulate the priorities of all items belonging to a related group.

We design a hyperbolic caching variant for several production systems from leading cloud providers. We implement our scheme in Redis and the Django web framework. Using real and simulated traces, we show that hyperbolic caching reduces miss rates by ~10-20% over competitive baselines tailored to the application, and improves end-to-end throughput by ~5-10%.

1 Introduction

Web applications and services aggressively cache data originating from a backing store, in order to reduce both access latency and backend load. The wide adoption of Memcached [23] and Redis [44] (key-value caching), Guava [26] (local object caching), and Varnish [50] (front-end HTTP caching) speak to this demand, as does their point-and-click availability on cloud platforms like Heroku via MemCachier [38], EC2 via ElastiCache [4], and Azure via Azure Redis Cache [7].

Caching performance is determined by the workload and the *caching algorithm*, i.e., the strategy for prioritizing items for eviction when the cache is full. All of the above services employ inflexible caching algorithms, such as LRU. But the needs of each application vary, and significant performance gains can be achieved by tailoring the caching strategy to the application: e.g., incorporating cost of fetching, expiration time, or other factors [8, 46]. Function-based strategies [2, 52] take this approach, by devising functions that combine several of these factors.

All of these strategies are fundamentally limited, however, because they rely on data structures (typically priority queues) to track the ordering of cached items. In particular, an item's priority is only changed when it is accessed. However, does cache eviction need to be tied to a data structure? Caches like Redis already eschew ordering data structures to save memory [45]. Instead, they rely on random sampling to evict the approximately lowest-priority item [42]: a small number of items are sampled from the cache, their priorities are evaluated (based on per-item metadata), and the item with lowest priority is evicted. Can this lack of an ordering data structure enable us to build a caching framework with vast flexibility? Indeed, we show that the combination of *random sampling* and *lazy evaluation* allows us to evolve item priorities arbitrarily; thus we can freely explore the design space of priority functions! Neither Redis nor existing algorithms exploit this approach, yet we find it outperforms many traditional and even domain-optimized algorithms.

Armed with this flexibility, we systematically design a new caching algorithm for modern web applications, called *hyperbolic caching* (§2). We begin with a simple theoretical model for web workloads that leads to an optimal solution based on frequency. A key intuition behind our approach is that caches can scalably measure item frequency *only while items are in the cache*. (While some algorithms, e.g., ARC [37], employ *ghost caches* to track items not in the cache, we focus on the more practical setting where state is maintained only for cached items.) Thus, we overcome the drawbacks of prior frequency-based algorithms by incorporating the time an item spends in the cache. This deceptively simple modification already makes it infeasible to use an ordering data structure, as pervasively employed today, because item priorities decay at variable rates and are continuously being reordered. Yet with hyperbolic caching, we can easily customize the priority function to different scenarios by adding *extensions*, e.g., for item cost, expiration time, and windowing (§3). We also introduce the notion of *cost classes* to man-

age groups of related items, e.g., items materialized by the same database query. Classes enable us both to more accurately measure an item's miss cost (by averaging over multiple items) and to adjust the priorities of many items at once (e.g., in response to a database overload).

A quick survey of existing algorithms shows that they fall short of this flexibility in different ways. Recency-based algorithms like LRU use time-of-access to order items, which is difficult to extend: for example, incorporating costs requires a completely new design (e.g., GreedyDual [53]). Frequency-based algorithms like LFU are easier to modify, but any non-local change to item priorities—e.g., changing the cost of multiple items—causes expensive churn in the underlying data structure. Some algorithms, such as those based on marking [22], maintain only a partial ordering, but the coarse resolution makes it harder to incorporate new factors. Several theoretical studies [2,46] formulate caching as an optimization problem unconstrained by any data structure, but their solutions are approximated by online heuristics that, once again, rely on data structures.

We design a hyperbolic caching variant for several different production systems from leading cloud providers (§3), and evaluate them on real traces from those systems. We implement hyperbolic caching in Redis and the Django web framework [18], supporting both per-item costs and cost classes (§4). Overall (§5), we find that hyperbolic caching reduces miss rates by ~10-20% over competitive baselines tailored to the application, and improves end-to-end system throughput by ~5-10%. This improvement arises from changing only the caching algorithms used by existing systems—our modification to Redis was 380 lines of code—and nothing else.

To summarize, we make the following contributions:

1. We systematically design a new caching algorithm for modern web applications, hyperbolic caching, that prioritizes items in a radically different way.
2. We define extensions for incorporating item cost and expiration time, among others, and use them to customize hyperbolic caching to three production systems.
3. We introduce the notion of cost classes to manage groups of related items effectively.
4. We implement hyperbolic caching in Redis and Django and demonstrate performance improvements for several applications.

Although we only evaluate medium-to-large web applications, we believe hyperbolic caching can improve hyper-scale applications like Facebook, where working sets are still too large to fit in the cache [6, 49].

2 Hyperbolic Caching

We first describe the caching framework required by hyperbolic caching (§2.1). Then, we motivate a simple theoretical model for web workloads and show that a classical frequency approach is optimal in this model (§2.2). By solving a fundamental challenge of frequency-based caching (§2.3), we arrive at hyperbolic caching (§2.4).

2.1 Framework

We assume a caching service that supports a standard `get`/`put` interface. We make two changes to the implementation of this interface. First, we store a small amount of metadata per cached item i (e.g., total number of accesses) and update it during accesses; this is done by the `on_get` and `on_put` methods in Fig. 1. Second, we remove any data structure code that was previously used to order the items. We replace this with a priority function `p(i)` that maps item i's metadata to a real number; thus `p` imposes a total ordering on the items. To evict an item, we randomly sample S items from the cache and evict the item i with lowest priority `p(i)`, as implemented by `evict_which`. This approximates the lowest-priority item [42]; we evaluate its accuracy in §5.3.

The above framework is readily supported by Redis, which already avoids ordering data structures and uses random sampling for eviction. The use of metadata and a priority function is standard in the literature and referred to as "function-based" caching [8]. What is different about our framework is *when* this function is evaluated. Prior schemes [2, 46, 52] evaluate the function on each `get`/`put` and use the result to (re)insert the item into a data structure, freezing its priority until subsequent accesses. Our framework uses *lazy evaluation* and no data structure: an item's priority is only evaluated when it is considered for eviction, and it can evolve arbitrarily before that point without any impact on performance.

2.2 Model and frequency-based optimality

In many workloads, the requests follow an item popularity distribution and the time between requests for the same item are nearly independent [10]. Absent real data, most systems papers analyze such distributions (e.g., Zipfian [20, 56]), and model dynamism as gradual shifts between static distributions. Motivated by this, we model requests as a *sequence of static distributions* $\langle D_1, D_2, \ldots \rangle$ over a universe of items, where requests are drawn independently from D_1 for some period of time, then from D_2, and so on. The model can be refined by constraining the transitions (D_i, D_{i+1}), but even if we assume they are instantaneous, we can still prove some useful facts (summarized below). Our measure of cost is the miss rate, which is widely used in practice.

```
def evict_which():
  sampled_items = random_sample(S)
  return argmin(p(i) for i in sampled_items)

def on_put(item):
  item.accessed = 1
  item.ins_time = timenow()
  add_to_sampler(item)

def on_get(item):
  item.accessed += 1

def p(item):
  in_cache = timenow - item.ins_time
  return item.accessed / (in_cache)
```

Figure 1: Pseudocode for hyperbolic caching in our framework.

Within a distribution D_i, a simple application of the law of large numbers shows that the optimal strategy for a cache of size k is to cache the k most popular items. This is closely approximated by the least-frequently-used (LFU) algorithm: a typical implementation assigns priority n_i/H to item i, where n_i is the number of hits to i and $H = \sum_i n_i$ is the sum over all cached items. Whereas LFU approximates the optimal strategy, one can prove that LRU suffers a gap. This is in contrast to the traditional competitive analysis model—which assumes a *worst-case* request sequence and use *total misses* as the cost [47]—in which LRU is optimal. This model has been widely criticized (and improved upon) for being pessimistic and unrealistic [3, 9, 32, 33, 54]. Our model is reminiscent of older work (e.g., [24]) that studied independent draws from a distribution but, again, used total misses as the cost.

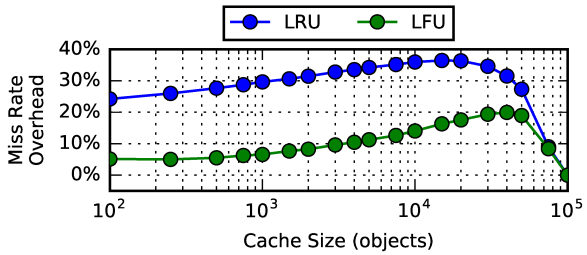
To validate our theoretical results, we use a static Zipfian popularity distribution and compare the miss rates of LRU and LFU to the optimal strategy, which has perfect knowledge of every item's popularity (Fig. 2).[1] Until the cache size increases to hold most of the universe of items, LRU has a 25-35% higher miss rate than optimal. LFU fares considerably better, but is far from perfect. We address the drawbacks of LFU next.

2.3 Problems with frequency

Even if requests are drawn from a stable distribution, there will be irregularities in practice that cause well-known problems for frequency-based algorithms:

New items die. When an item is inserted into the cache, the algorithm does not have a good measure of it's popularity. In LFU, a new item gets a frequency count of 1, and may not have enough time to build up its count to survive in the cache. In the worst case, it could be repeatedly inserted and evicted despite being requested frequently.

[1]We present miss rate rather than hit rate curves because our focus is on the penalties at the backend. Higher numbers indicate *worse* performance in most figures, and the last datapoint is 0 because the cache is large enough to never incur a miss.



Cache Size	3k	10k	30k	100k
Perfect Freq. Miss Rate	0.29	0.19	0.10	0.00

Figure 2: Simulated miss rates[1] compared to a strategy with perfect frequency knowledge. Items are sampled with Zipfian popularity ($\alpha \approx 1$) from 10^5 items. The cache is configured to hold a fixed number of objects (rather than simulating size in bytes).

This problem can be mitigated by storing metadata for non-cached items (e.g., [37]), but at the cost of additional memory that is worst-case linear in the universe size.

Old items persist. When items' relative popularities shift—e.g., moving from D_i to D_{i+1} in our model—a frequency approach may take time to correct its frequency estimates. This results in older items persisting in the cache for longer than their current popularity warrants. For example, consider a new item with 1 access and an older item with 2 accesses. Initially, the new item may be better to cache, but if time passes without an additional access, our knowledge of the old item is more reliable.

2.4 Hyperbolic Caching

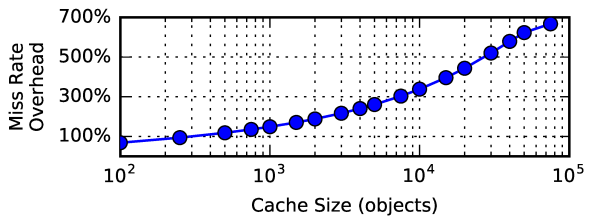
We solve the above problems by incorporating a *per-item* notion of time. Intuitively, we want to compensate for the fact that caches can only measure the frequency of an item *while it is in the cache*. Traditional LFU does not account for this, and thus overly punishes new items.

In our approach, an item's priority is an estimate of its frequency since it entered the cache:

$$p_i = \frac{n_i}{t_i} \tag{1}$$

where n_i is the request count for i since it entered the cache and t_i is the time since it entered the cache. This state is erased when i is evicted. Fig. 1 provides pseudocode for this policy, which we call *hyperbolic caching*.

Hyperbolic caching allows a new item's priority to converge to its true popularity from an initially high estimate. This initial estimate gives the item temporary immunity (similar to LRU), while allowing the algorithm to improve its estimate of the item's popularity. Over time, the priority of each item drops along a hyperbolic curve. Since each curve is unique, the ordering of the items is continuously changing. Such reordering is uniquely enabled by our framework (lazy evaluation, random sampling), and



Cache Size	3k	10k	30k	100k
HC Miss Rate	0.30	0.21	0.13	0.00

Figure 3: LFU miss rate compared to hyperbolic caching (HC) on a dynamic Zipfian workload ($\alpha \approx 1$), where new items are introduced into the distribution every 100 requests.

would be very costly to implement with a data structure.[2]

The strengths of hyperbolic caching over LFU are readily apparent in workloads that slowly introduce new items into the request pool. Fig. 3 shows that LFU has a significantly higher miss rate on a workload that introduces new items every 100 requests whose popularities are in the top 10% of a Zipfian distribution. This workload is artificial and much more dynamic than we would expect in practice, but serves to illustrate the difference.

Another way to solve the same problem is to multiplicatively degrade item priorities (e.g., LRFU [34]) or periodically reset them. Both of these are forms of windowing, which best addresses the problem of old items persisting, *not* the problem of new items dying. We compare hyperbolic caching to these approaches in §3.4.

3 Customizing Hyperbolic Caching

Our framework allows us to build on the basic hyperbolic caching scheme by adding *extensions* to the priority function and storing metadata needed by those extensions. This is similar to the way function-based policies build on schemes like LRU and LFU [2, 46, 52], but in our case the extensions can freely modify item priorities without affecting efficiency (beyond the overhead of evaluating the function). Which extensions to use and how to combine them are important questions that depend on the application. Here, we describe several extensions that have benefited our production applications (cost, expiration time) and our understanding of hyperbolic caching's performance (windowing, initial priority estimates).

3.1 Cost-aware caching

In cost-aware caching, all items have an associated cost that reflects the penalty for a miss on the item. The goal is to minimize the total cost of all misses. Cost awareness is particularly relevant in web applications, because unlike traditional OS uses of caching (fixed-size CPU instruction lines, disk blocks, etc.), the cost of fetching different items can vary greatly: items vary in size, can originate from different backing systems or stores, or can be the materialized result of complex database joins.

Much of the prior work on cost-aware caching focuses on adapting recency-based strategies to cost settings (e.g., GreedyDual [11]). This typically requires a new design, because recency-based strategies like LRU-K [41] and ARC [37] use implicit priorities (e.g., position in a linked list) and metrics like time-of-access, which are difficult to augment with cost. In contrast, frequency-based approaches like hyperbolic caching use explicit priorities that can naturally be multiplied by a cost: $p'_i = c_i p_i$, where c_i is the cost of fetching item i and p_i is the original (cost-oblivious) priority of i. Note that p_i may include other extensions from later sections.

The cost of an item needs to be supplied to the caching algorithm by the application. It can take many forms. For example, if the goal is to limit load on a backing database [35], the cost could be request latency. If the goal is to optimize the hit rate per byte of cache space used, the cost could be item size [11].

Real-world applications. Our evaluation studies two applications which benefit from cost awareness. The first is a set of applications using Memcachier [38], a production cloud-based caching service built on Memcache. We use costs to account for object size in the eviction decision, i.e., set $c_i = 1/s_i$ where s_i is the size of item i. The second application is Viral Search [25, 51], a Microsoft internal website that displays viral stories from Twitter in tree form. Virality is measured by analyzing the diffusion tree of the story as it is shared through the network. For each story, the website fetches the tree edges and constructs and lays them out for display. The final trees are cached and the cost of each is set to the time required to construct and lay out the tree.

3.2 Cost classes

In many applications, the costs of items are related to one another. For example, some items may be created by database joins, while others are the result of simple indexed lookups. Rather than measuring the cost of each item individually, we can associate items with a *cost class* and measure the performance of each class. We store a reference to the class in each item's metadata.

Cost classes have two main advantages. Consider the example of request latency to a backend database. If costs are maintained per item, latencies must be measured for each insertion into the cache. Since these measurements are stochastic, some requests will experience longer de-

[2]The basic hyperbolic function in Eq. 1 can be tracked by a kinetic heap [31], but this is a non-standard structure with $O(\log^2 n)$ update time, and it ceases to work if the extensions from §3 are added.

lays than others and thus be treated as more costly by the cache, even though the higher cost has nothing to do with the item itself. What's more, the higher costs will keep these items in the cache longer, preventing further updates because costs are only measured when a miss occurs. By using cost classes, we can aggregate latency measurements across all items of a class (e.g., in a weighted moving average), resulting in a less noisy estimate of cost.

The second advantage of cost classes comes from longer-term changes to costs. In scenarios where a replica failure or workload change affects the cost of fetching a whole class of items, existing approaches would only update the individual costs after the items have been evicted, one by one. However, when using cost classes, a change to a class's cost is immediately applied to both newly cached items and items *already* in the cache.

In both cases above, a single update to a cost class changes the priorities of many items at once, possibly dramatically. Our framework supports this with little additional overhead because 1) items store a *reference* to the class information, and 2) priorities are lazily evaluated. In contrast, integrating cost classes into existing caching schemes is prohibitively expensive because it incurs widespread churn in the data structures they rely on.

Interestingly, some production systems already employ cost classes implicitly, via more inflexible and inelastic means. For example at Facebook, the Memcached service is split among a variety of pools, such that keys that are accessed frequently but for which a miss is cheap do not interfere with infrequently accessed keys for which a miss is very expensive [40]. However, this scheme requires much more management and requires tuning pool sizes; more importantly, it does not automatically adapt to changes in request frequencies or item costs.

In our experiments, we implement cost classes using exponentially weighted moving averages. We explored other techniques such as non-weighted moving averages and using the most recent cost, but exponentially weighted moving averages performed the best on our workloads while requiring little memory overhead for tracking.

While cost classes are useful in many settings, incorrectly assigning objects to the same class that do not share the same underlying cost will degrade caching performance. In some settings, objects may be members of multiple classes concurrently—there are several ways of handling this, but we do not explore this in our work.

Real-world application. Django is a Python framework for web apps that includes a variety of libraries and components. One such component adds support for whole-page caching. We modified this middleware to support cost awareness, as follows. In Django, page requests are dispatched to "view" functions based on the URL. We associate a cost class with each view function, and map individual pages to their view function's class.

3.3 Expiration-aware caching

Many applications need to ensure that the content conveyed to end users is not stale. Developers achieve this by specifying an *expiration time* for each item, which tells the caching system how long the item remains valid. While many systems support this feature, it is typically handled by an auxiliary process that has no connection to the caching algorithm (apart from evicting already-expired items). But incorporating expiration into caching decisions makes intuitive sense: if an item is going to expire soon, it is less costly to evict than a similarly popular item that expires later (or not at all).

To add expiration awareness to hyperbolic caching, we need to strike a balance between the original priority of an item and the time before it expires. Rather than evict the item least likely to be requested *next*, we want to evict the item most likely to be requested *the least number of times over its lifetime*. This can be naturally captured by multiplying item i's priority by the time remaining until expiry, or $\max((t_{exp_i} - t_{cur}), 0)$. However, this scheme equally prioritizes requests far into the future and those closer to the present, which is unideal because estimates about the future are less likely to be accurate (e.g., the item's popularity may change). Therefore, instead of equally weighting all requests over time, we use a weighting function that discounts the value of future requests:

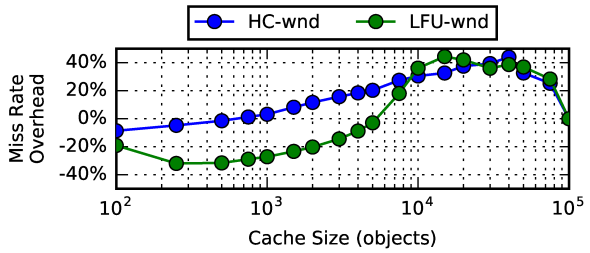
$$p_i' = p_i \cdot (1 - e^{-\lambda \cdot \max((t_{exp_i} - t_{cur}), 0)})$$

where p_i is the original (expiration-unaware) priority of item i and λ is a parameter controlling how quickly to degrade the value of future requests. As an item's time until expiration decreases, this weighting function sharply approaches zero. Thus the function continually reweights (reorders) item priorities, which is uniquely enabled by our framework: existing approaches can only account for expiration time once, on insertion into a data structure.

Real-world application. The Decision Service [1,39] is a machine learning system for optimizing decisions that has been deployed in MSN to personalize news articles shown to users. Given a user request, a particular article is featured and a reward signal (e.g., click) is recorded. Since rewards may arrive after a substantial delay, a cache is used to match the decision to its reward. Rewards are only valid if they occur within a time window after the decision, so each cached item is given an expiration time.

3.4 Windowing

Windowing is often used in frequency-based caching to adapt to dynamic workloads and address the problem of



Cache Size	3k	10k	30k	100k
HC Miss Rate	0.32	0.22	0.13	0.00
LFU Miss Rate	0.44	0.21	0.13	0.00

Figure 4: Adding perfect windowing to hyperbolic caching and LFU on a dynamic Zipfian workload ($\alpha \approx 1$). Each curve is compared to the algorithm's non-windowed performance (given in the table). The window size is fixed at 10^4 requests. Every 100 requests, an item is promoted to the top of the distribution.

"old items persisting". The idea is to forget requests older than a fixed time window from the present. Hyperbolic caching naturally achieves the benefits of windowing, but we investigate it for two reasons. First, one can show that hyperbolic caching, unlike LRU, is not optimal in the traditional competitive analysis model [47], but it can be made optimal if windowing is used. Second, windowing represents alternative solutions, such as resetting or multiplicatively degrading frequency estimates (e.g., LRFU [34]), and so serves as an informative comparison.

We simulate windowing using an idealized (but completely inefficient) scheme that tracks every request and forgets those older than the window. This upper bounds the potential gains of windowing. Fig. 4 shows the performance of LFU and hyperbolic caching on a dynamic Zipfian workload, with and without windowing. For hyperbolic caching, windowing provides limited benefits: 5–10% reduction in misses on small cache sizes; LFU benefits more but again on small cache sizes. The problem is that windowing discards measurements that help the cache estimate item popularity. Even in dynamic workloads, we find that large-sized caches can accommodate newly popular items, so performance depends more on the ability to differentiate at the long tail of old items. Fortunately, hyperbolic caching's measure of time in the cache achieves some of the benefits of windowing; it outperforms even recency-based approaches on many of the highly dynamic workloads we evaluated.

3.5 Initial priorities

Hyperbolic caching protects newly cached items by giving them an initial priority that tends to be an *overestimate*: for example, an item with true popularity of 1%—placing it among the most popular in most realistic workloads—would remain overvalued for at least 100 timesteps of hyperbolic decay. We found that adjusting the initial priority based on that of recently evicted items alleviates this problem, because evicted items tend to have similar priorities in the tail of the distribution. Thus, we set a new item's initial priority to a mixture of its original priority (p_i) and the last evicted item's priority (p_e): $p_i' = \beta p_i + (1-\beta)p_e$. Solving this for n_i in Eq. 1 gives us the initial request count to use, after which the extension can be discarded. β requires some tuning: we found that $\beta = 0.1$ works well on many different workloads; for example, on a Zipfian workload ($\alpha \approx 1$) it reduced the miss rate by between 1% and 10% over hyperbolic caching for all cache sizes.

4 Implementation

Our evaluation uses both simulation and a prototype implementation. For the simulations, we developed a Python application that generates miss rate curves for different caching strategies and workloads. For our prototype, we implemented hyperbolic caching in Redis and developed Django middleware that uses the modified Redis. Our code is open-source [28].

Redis. We modified Redis (forked at 3.0) to use the hyperbolic caching framework. This was straightforward because Redis already uses random sampling for eviction. We included support for per-item costs (and size awareness), cost classes tracked with an exponentially weighted moving average, and initial priorities. Excluding diagnostic code, this required 380 lines of C code.

We store the following metadata per item, using double-precision fields: item cost, request count, and time of entry (from Eq. 1 and §3.1). This is two doubles of overhead per item compared to LRU. Our prototype achieved similar miss rates to our simulations, suggesting this precision is adequate. Exploring the trade-offs of reduced precision in these fields is left to future work.

Django caching middleware. Django is a framework for developing Python web applications. It includes support for middleware classes that enable various functionality, such as the Django whole-page caching middleware. This middleware interposes on requests, checking a backend cache to see whether a page is cached, and if so, the content is returned to the client. Otherwise, page processing continues as usual, except that the rendered page is cached before returning to the client. We added middleware to track cost information for web pages; we measure cost as the CPU time between the initial miss for a page and the subsequent SET operation, plus the total time for database queries. This avoids time lost due to processor scheduling. We subclassed the Django Redis caching interface to convey cost information to our Redis implementation. The interface supports caching a page

with/without costs, and optionally specifying a cost class for the former. Cost classes are associated with the particular Django "view" function that renders the page. In total, this was implemented in 127 lines of Python code.

5 Evaluation

Our evaluation explores the following questions:

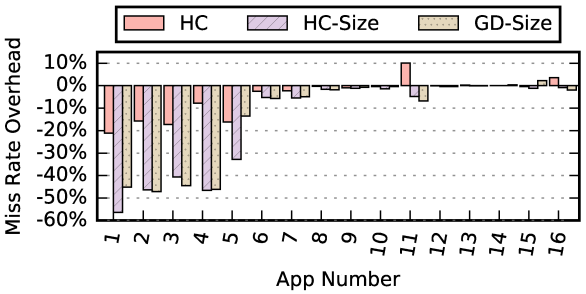
1. How does hyperbolic caching compare to current caching techniques in terms of miss rate?
2. Does our implementation of hyperbolic caching in Redis improve the throughput of web applications?
3. What effect does sample size have on the accuracy and performance of our eviction strategy?

We use real application traces (§5.1) and synthetic workloads designed to emulate realistic scenarios (§5.2). We evaluate these questions using simulations as well as deployments of Django and NodeJS, using our prototype of hyperbolic caching in Redis. To drive our tests, our applications run on Ubuntu 14.04 servers located on a single rack with Intel Xeon E5620 2.40GHz CPUs. Applications use PostgreSQL 9.3 as their backing database. For throughput tests, our systems were loaded exclusively by the test, and to measure max throughput, we increased the rate of client requests until throughput plateaued and the application server experienced 100% CPU load.

Methodology. For the majority of our standard workloads, we use a Zipfian request distribution with $\alpha \approx 1$. This is the same parameterization as many well-studied benchmarks (e.g., YCSB [15]), though some like linkbench [5] use a heavier-tailed $\alpha = 0.9$. When measuring miss rates, we tally misses after the first eviction (i.e., we allow the cache to fill first). For workloads with associated item costs, misses are scaled by cost. For real traces, we run the tests exactly as prescribed; for workloads based on popularity distributions, we generate enough requests to measure the steady state performance. When choosing a cache size to compare performance amongst algorithms, we use the size given by the trace, or if not given we use sizes corresponding to high and middle range hit rates (roughly 90% and 70%), which reflect the cache hit rates reported in many deployed settings (e.g, [6, 27]). In Facebook [27], of the 35.5% of requests that leave a client's browser (the rest are cached locally), ~70% are cached in either the edge cache or the origin cache. For our random sampling, unless otherwise noted, we sample 64 items.

5.1 Real-world workloads

We evaluate real applications in two ways. When lacking access to the actual application code or deployment setting, we evaluate the performance through simulation. For other applications, we measure the performance using our prototype implementation of Django caching paired with Redis. The applications below were described in §3, when we customized hyperbolic caching to each one.



(a) Simulated miss rates compared to the miss rate of LRU.

App Number	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Mean Obj. Sz. (kB)	79.9	15.4	1.8	149.7	561.7	2.3	1.1	25.0
Stdev Obj. Sz. (kB)	116.5	40.4	9.0	254.5	100.8	3.1	7.9	46.6

App Number	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*
Mean Obj. Sz. (kB)	5.4	8.1	7.2	5.5	2.2	30.8	26.0	9.0
Stdev Obj. Sz. (kB)	5.1	13.5	17.9	2.0	4.3	52.2	3.1	25.4

(b) Means and stdevs. of object sizes in app traces.

Figure 5: Caching performance on Memcachier app traces.

5.1.1 Memcachier applications (from §3.1)

To evaluate the Memcachier applications, we processed a trace of GET and SET requests spanning hundreds of applications, using the amount of memory allocated by each application as the simulated cache size. We focused our attention on the 16 applications with over 10k requests whose allocation could not fit all of the requested objects (many applications allocated enough memory to avoid any evictions). We measured the miss rates of plain HC and LRU, and then used the object sizes to evaluate our size-aware extension, HC-Size, and the GD-Size [11] algorithm. Fig. 5 show the performance of the algorithms over a single execution of each application's trace.

In our evaluation, HC outperforms LRU in many applications, and HC-Size drastically outperforms LRU. While GD-Size is competitive with HC-Size, our framework allows for the implementation of HC-Size with only two lines of code, whereas implementing GD-Size from LRU requires an entirely new data structure [11].

5.1.2 Decision Service (from §3.3)

The Decision Service [1, 39] is a machine learning system for optimizing decisions that has been deployed in MSN. The service uses a cache to join information about each decision with the corresponding reward signal. Because rewards must be received within a given period of time, information is cached with an expiration time.

	Decision Service	Viral Search
Algo.	Miss Rate (Δ%)	Miss Rate (Δ%)
HC	0.60 (+0%)	0.17 (+0%)
HC-Expire	0.55 (-8%)	—
LRU/GD	0.55 (-8%)	0.18 (+6%)
ARC	0.55 (-8%)	0.22 (+29%)
LFU	0.99 (+65%)	0.16 (-6%)
Cache Size	1k	35k

Figure 6: Simulated performance on real-world traces.

Cache Alg.	Miss Rate (Δ)	Tput. (Δ)
Default	0.681 (+0.0%)	21.1 req/s (+0.0%)
HC	0.637 (-6.5%)	23.7 req/s (+12.3%)
HC-Cost	0.669 (-1.8%)	21.1 req/s (+0.0%)
HC-Class	0.639 (-6.2%)	22.6 req/s (+7.1%)
Obj. Sizes	$\mu = 32.0$kB	$\sigma = 31.6$kB

Figure 7: Performance of Django-Wiki application using HC Redis compared to default Redis. The cache is sized to 1GB and the workload is a 600k trace of Wikipedia article requests.

In this workload, because items have the same expiration time and are accessed only once after insertion (to join the reward information), recency is roughly equal to time-until-expiration. Therefore, LFU and HC perform poorly in comparison to a recency strategy (Fig. 6). However, our expiration-aware extension allows HC-Expire to perform just as well as the recency strategies.

5.1.3 Viral Search (from §3.1)

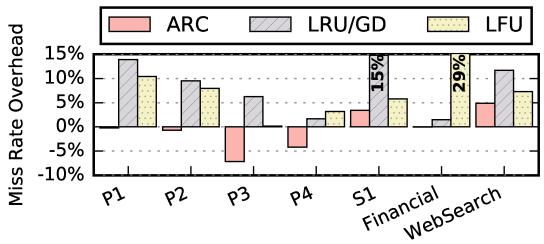
The Viral Search [25,51] application is an interactive website that displays viral stories from a large social network. Each viral story is represented as a tree that requires varying amounts of time to construct and layout on the server side. We use this time as the per-item cost and apply our cost-aware extension. Items are requested based on a popularity distribution given by each item's "virality score" and we measure performance over 10M requests.

Hyperbolic caching performs well on this cost-aware workload, beating all algorithms except for LFU (Fig. 6), and suffering 6% fewer misses than GreedyDual.

5.1.4 Django Wiki application (from §3.2)

We evaluate our caching scheme on an open-source Django wiki app using our Django caching middleware. The caching middleware stores cached data using a configurable backend, for which we use either the default Redis or our modified version with hyperbolic caching.

The wiki database serves a full copy of articles on Wikipedia from Jan. 2008. We measured the throughput and miss rate of the application using a trace of Wikipedia article requests from Sept. 1, 2007 (Fig. 7). We see an improvement in both miss rate and throughput when using HC rather than default Redis. Note that because the pages are costly to render, even small improvements in miss rate increase the throughput of the application. For this application, requests are only processed by two different Django views.



Trace	P1	P2	P3	P4	S1	F	WS
Cache Sz (*objs*)	32k	32k	32k	32k	525k	32k	525k
Miss Rate	0.72	0.73	0.88	0.91	0.81	0.50	0.85

Figure 8: Miss rates compared to HC on traces from the ARC paper and SPC (HC's miss rates are in the table). Cache sizes chosen based on sizes given in the ARC paper.

However, using HC-Cost *reduces* the system throughput compared to HC. This is because the time to render a page is similar across most pages, but has high variance: for one page, the mean time of fifty requests was 570ms with a deviation of 180ms. This leads a cost-aware strategy to incorrectly favor some pages over others. HC-Class alleviates this by reducing some of the variance, but it still performs worse than the cost-oblivious HC. For this application, using costs is counter-productive.

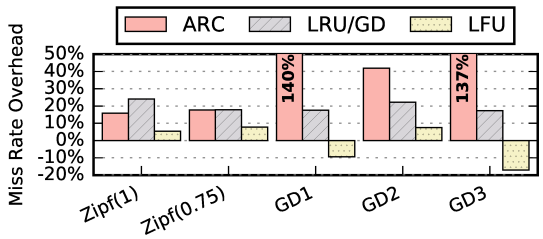
5.1.5 ARC and SPC traces

We additionally simulate performance on traces from ARC [37] and SPC [48] (Fig. 8). The P1-4 traces are memory accesses from a workstation computer; S1 and WebSearch are from a server handling web searches; and the Financial workload is an OLTP system trace. Caches were sized according to the ARC paper, and these sizes were used for the SPC traces as well. These traces have very high miss rates on all eviction strategies. However, HC performs very well, outperforming LRU in every workload and underperforming ARC in the P1-4 traces only. Importantly, on workloads where LFU exhibits poor performance, HC remains competitive with ARC, demonstrating the effectiveness of our improvements over LFU.

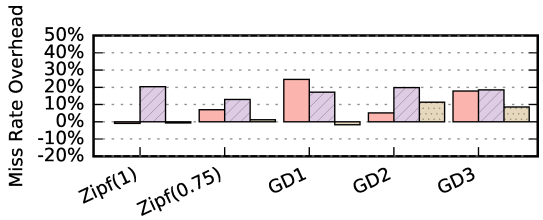
5.2 Synthetic workloads

In this section, we simulate and compare the performance of HC to three popular strategies—ARC, LFU, and LRU—on synthetic workloads that reflect the demands of today's caches. For cost-aware workloads, we extend LRU with GreedyDual, and we modify LFU by multiplying frequencies by cost. (ARC is not amenable to costs.)

For each synthetic workload, we evaluate the performance of each caching algorithm on two cache sizes, corresponding to a 90% and a 70% hit rate with hyper-



(a) Cache size fixed where hit rate of HC $\approx 90\%$.



(b) Cache size fixed where hit rate of HC $\approx 70\%$.

Figure 9: Miss rates on synthetic workloads with 10M requests. Miss rates are compared to the performance of HC. For cost-aware strategies (GD1-GD3), misses are scaled by the cost of the missed item.

bolic caching (Fig. 9). Note that while we simulated relatively small key spaces, we evaluated our Redis prototype on larger key spaces and found similar improvements in miss rate and overall system throughput. In general, these workloads suggest that HC can perform very well in a variety of scenarios.

The most striking improvement relative to ARC is on workloads GD1-3. These workloads have associated costs and are based on the workloads described in GDWheel [35]. Since ARC is a cost-oblivious strategy, it does poorly on these workloads. However, even in workloads without cost, our scheme is competitive with ARC.

5.2.1 Synthetic web application performance.

In order to understand how our improved miss rates affect end-to-end throughput in modern web servers, we configured a NodeJS web app to use a backing database with Redis as a look-aside cache. We drive HTTP GET requests to the web app from a client that draws from synthetic distributions. The web app parses the URL and returns the requested object. Objects are stored as random 32B strings in a table with object identifier as the primary key.

Relating cache misses to throughput. To understand the association between miss rate and throughput, we scaled the size of our Redis cache to measure system throughput with different miss rates (Fig. 10). Miss rate has a direct impact on throughput even when many client requests can be handled concurrently. Misses not only cause slower responses from the backend (an effect which can be mitigated with asynchronous processing), but they also require additional processing on the web server—on a miss, the app issues a failed GET, a SQL SELECT, and then a PUT request. This adds a direct overhead to the throughput of the system.

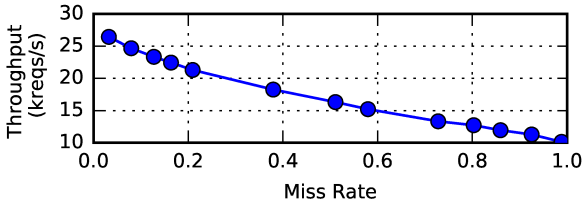


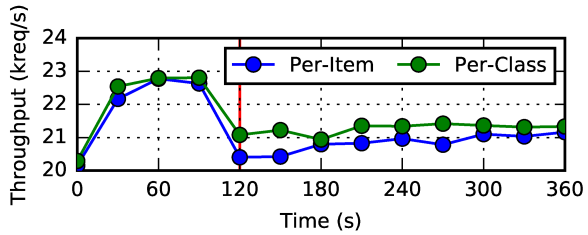
Figure 10: Throughput of NodeJS using Redis as a look-aside cache for PostgreSQL as the miss rate varies.

Cache sz. *(objs.)*	Default Redis Mean tput. *(kreq/s)*	Miss rate	HC Redis Mean tput. *(kreq/s)*	Miss rate	Δ tput.
Zipfian ($\alpha \approx 1, N = 10^5$)					
39k	18.1 ± 0.22	0.11	20.2 ± 0.18	0.09	10.3%
3k	9.1 ± 0.09	0.38	10.5 ± 0.06	0.31	13.5%
Zipfian ($\alpha = 0.75, N = 10^6$)					
125k	7.5 ± 0.06	0.55	7.7 ± 0.16	0.49	3.2%
70k	6.8 ± 0.06	0.64	7.3 ± 0.12	0.56	6.3%
Zipfian ($\alpha \approx 1, N = 10^6$)					
200k	14.6 ± 0.16	0.17	15.3 ± 0.13	0.16	4.4%
50k	11.2 ± 0.11	0.28	12.1 ± 0.20	0.24	7.1%
Dynamic Intro. ($N = 10^5$)					
42k	19.3 ± 0.17	0.10	20.6 ± 0.16	0.09	6.3%
5k	10.0 ± 0.15	0.33	11.3 ± 0.12	0.27	11.6%

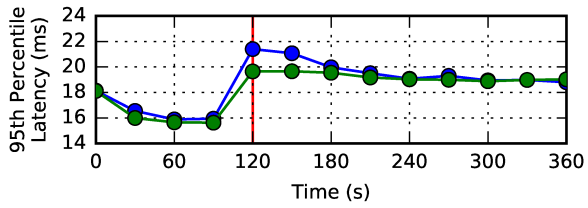
Figure 11: Miss rate and throughput of workloads running on NodeJS with a Redis cache. Each configuration was executed 10 times with workloads of 5M requests to objects of size 96B.

Zipfian distribution. We measured the maximum throughput of our NodeJS server when servicing requests sampled from synthetic workloads with zipfian request distributions (Fig. 11.) Depending on the workload, hyperbolic caching outperforms Redis's default caching algorithm (LRU approximated by random sampling) in miss rates by 10-37%, and improves throughput by up to 14% on some workloads. While throughput differences of 5-10% on some workloads may be modest, they are not insignificant, and come with little implementation burden.

Cost-aware caching. To measure the potential throughput benefits of cost-aware caching, we wrote a NodeJS app that makes two types of queries to the backend: (1) a simple key lookup and (2) a join. The app measures the latency of backend operations and uses that as the item's cost. In our experiment, the cache can hold 30k objects, and we drive the app with 1M requests sampled from a Zipfian distribution ($\alpha \approx 1$). When using normal HC, we measured a throughput of 5.0 kreq/s and a miss rate of



(a) Throughput measured over 30 second windows.



(b) Tail latency measured over 30 second windows.

Figure 12: Performance of NodeJS app fetching items from two different PSQL servers using HC with per-item and per-class costs. After 2 minutes, one PSQL server is stressed and takes longer to fetch items. The cache holds 30k objects and requests are Zipfian ($\alpha \approx 1$).

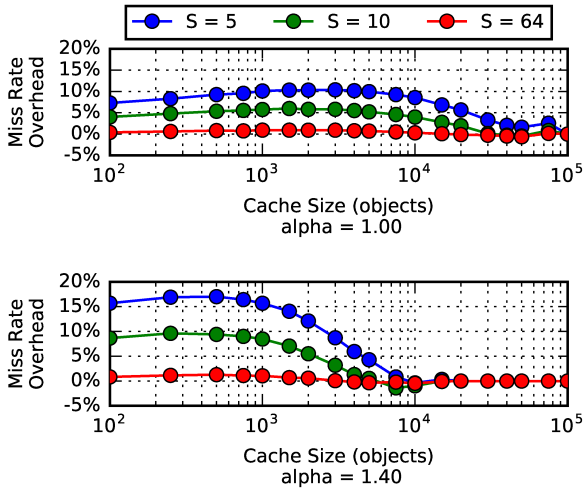


Figure 13: Simulated performance of HC for different sampling sizes compared to finding the true minimum. The request workloads are Zipfian distributions with different skew parameters.

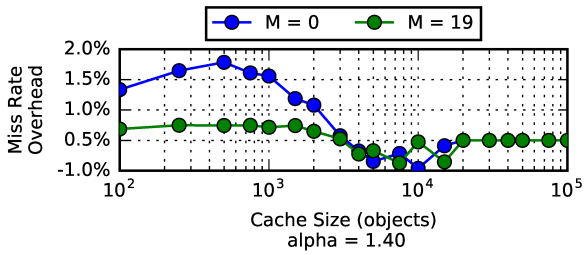


Figure 14: Simulation of HC using sampling technique that retains M items [42] on a Zipfian workload with $\alpha \approx 1.4$, compared to the performance of finding the true minimum.

0.11. When using HC-Cost, the miss rate was 0.17, which is 57% *higher*, but the throughput was 9.4 kreq/s, an 85% improvement over HC. HC-Cost traded off miss rate for lower overall cost, increasing overall performance.

Responding to backend load with classes. To demonstrate how cost classes can be used to deal with backend load, we designed a NodeJS application which performs key lookups on one of two different PSQL servers. The application measures the latency of the backend operation and uses that as the cost in our Redis prototype. Additionally, it sets the class of each cached object to indicate which backend served the object. This way, HC-Class will use a per-class cost estimate (exponentially WMA) when deciding which items to evict, rather than per-item. We evaluate the application by driving it with requests and measuring throughput and tail latency (Fig. 12). Two minutes into our test, we stress one PSQL backend using the Unix script `stress`. When one backend is loaded, throughput decreases and tail latency increases. By using per-class costs, HC-Class quickly adjusts to one class being more costly. With per-item costs, however, HC-Cost is only able to update the costs of items when they are (re)inserted. As a result, HC-Cost needs more time to settle to steady state performance as item costs are slowly updated to their correct values.

5.3 Accuracy of random sampling

Our eviction strategy's sampling impacts its miss-rate. Prior work [42] has studied the impact of this sampling in detail. Using order statistics [17], one can easily show that the expected rank of an evicted item is $n/(S+1)$, where n is the number of items in the cache and S is the sample size. For example, a cache of $n = 10$k items and a sample of $S = 64$ would evict the 154th lowest item on average. In practice we found that this loss of accuracy is not problematic. Specifically, we measured and compared the miss rate curves for varying sample sizes on two different popularity skews (Fig. 13). While the smoothness of the priority distribution impacts this accuracy—and extensions like expiration may introduce jaggedness into priorities—the dominating factor is how heavy the tail is and the likelihood of sampling an item from it. Sampling performs worse on the lighter-tailed distribution because there are fewer tail items in the cache, making them less likely to be sampled. However, for the sample size we use ($S = 64$), the performance gap relative to full accuracy is slight. Although this varies depending on the workload and cache-size, a sample of 64 items was large enough in all of our experiments, so the additional improvement of better sampling techniques would be limited. Further increasing the

sample size is not without cost: each sampled item's priority must be evaluated, which could become expensive depending on the complexity of the priority function.

Psounis and Prabhakar [42] proposed an optimization to random sampling that retains some number of samples between evictions. This can boost the accuracy of random sampling, however in our tests we found the miss rate benefits to be minimal. On the light-tail distribution (Fig. 14), we compare performance to the suggested settings of their technique. While performance does improve for smaller caches, the benefits are more limited as cache size increases. We believe this is because tail items in a large cache tend to be new items that are less likely to be retained from prior evictions, though a more in-depth analysis is needed to confirm this. As the benefits are limited (and parameters are sensitive to cache size and workload), we did not use this optimization.

6 Related Work

Our introduction and subsequent discussions survey the landscape of caching work, including recency-based approaches (e.g., [16, 41, 53]), frequency-based or hybrid approaches (e.g., [34,37]), marking algorithms and partial orderings (e.g., [16, 22]), and function-based approaches (e.g., [2, 46, 52]). All of these approaches rely on data structures and thus cannot achieve the flexibility and extensibility of hyperbolic caching.

Consider the approaches that improve recency caching by using multiple queues to incorporate some frequency measures into eviction. LRU-K [41] stores items in k queues and evicts based on the k-th most recent access. Other works employing multiple queues include 2Q [30], MQ [55], and LIRS [29]. ARC [37] automatically tunes the queue sizes of an LRU-2-like configuration. Several of these algorithms incorporate *ghost caches*, which track information about items no longer in the cache. (This technique could also be applied to hyperbolic caching, but we focused our work on caches that store information about items residing in the cache, as most production caches do.) All of these strategies incorporate frequency to balance the downsides of LRU. However, they are difficult to adapt to handle costs or other factors, due to their use of time-of-access metrics and priority orderings.

GreedyDual [53] exemplifies this difficulty because it attempts to incorporate cost into LRU, requiring a redesign. Cao and Irani [11] implemented GreedyDual using priority queues for size-aware caching in web proxies, and GDWheel [35] implemented GreedyDual in Memcached using a more efficient wheel data structure. The RIPQ system uses size awareness in a flash-based caching system [49]. Other cost-aware strategies have incorporated properties such as freshness (e.g., [46]), which is similar to expiration times but not as strict. In contrast to these approaches, a priority function based on frequency can easily adopt cost, expiration, or other factors.

Hyperbolic caching learns from the above and adopts a function-based approach based on frequency. The GDSF [13] work incorporates frequency into their priority function, while Yang and Zhang [52] use a priority function that is also similar to ours. However, these strategies build their solution on GreedyDual by setting an item's cost equal to its priority. In our tests, we found that the interaction between GreedyDual's priority queue and this frequency led to poor performance (3-4x the miss rate of LRU). Moreover, using a queue forces these strategies to "freeze" an item's priority once it enters the structure; in contrast, our priorities evolve continuously and freely.

Recent work in the systems community has looked at other aspects of caching that we do not address, such as optimizing memory overheads [19, 21], multi-tenant caching [14, 43], balancing memory slabs [14], cache admission [19], and reducing flash erasures when using flash storage [12, 36, 49]. Hyperbolic caching does not require memory for ordering data structures, but uses space to store the metadata used to compute item priorities. We have not studied allocation across multiple caches, but note that our framework obviates the need for separately tuned caches in some cases, e.g., by using our cost class extension to manage the pools of caches described in [40].

7 Conclusion

We have presented the design and implementation of hyperbolic caching. Our work combines theoretical insights with a practical framework that enables innovative, flexible caching. Notably, the priority function we use reorders items continuously along hyperbolic curves. We implemented our work in Redis and Django and applied it to a variety of real applications and systems. By using different extensions, we are able to match or exceed the performance of one-off caching solutions. A deeper analysis of the described extensions, such as for cost classes and expiration times, is part of our future work.

Acknowledgments. This work was supported by NSF CAREER Award #0953197. Part of this work was conducted during an internship at MSR NYC. We thank Amit Levi for access to Memcachier traces and Asaf Cidon for his help in obtaining them. We thank Siddhartha Jayanti for his assistance with theoretical analyses. Muthu Muthukrishnan coined the name "hyperbolic caching". Finally, we thank our shepherd, Rachit Agarwal.

References

[1] A. Agarwal, S. Bird, M. Cozowicz, L. Hoang, J. Langford, S. Lee, J. Li, D. Melamed, G. Oshri, O. Ribas, S. Sen, and A. Slivkins. A multiworld testing decision service. *CoRR*, abs/1606.03966, 2016.

[2] C. C. Aggarwal, J. L. Wolf, and P. S. Yu. Caching on the world wide web. *IEEE Trans. Knowledge and Data Eng.*, 11(1):95–107, 1999.

[3] S. Albers, L. M. Favrholdt, and O. Giel. On paging with locality of reference. *J. Comput. Syst. Sci*, 70(2):145–175, 2005.

[4] Amazon ElastiCache. `http://aws.amazon.com/elasticache`.

[5] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: a database benchmark based on the Facebook social graph. In *SIGMOD*, 2013.

[6] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[7] Azure Redis Cache. `https://azure.microsoft.com/en-us/services/cache`.

[8] A. Balamash and M. Krunz. An overview of web caching replacement algorithms. *IEEE Communications Surveys and Tutorials*, 6(1-4):44–56, 2004.

[9] A. Borodin, S. Irani, P. Raghavan, and B. Schieber. Competitive paging with locality of reference. *J. Comput. System Sci.*, 50(2):244–258, 1995.

[10] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM*, pages 126–134, 1999.

[11] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proc. Symposium on Internet Technologies and Systems (USITS)*, 1997.

[12] Y. Cheng, F. Douglis, P. Shilane, M. Trachtman, G. Wallace, P. Desnoyers, and K. Li. Erasing belady's limitations: In search of flash cache offline optimality. In *USENIX ATC*, 2016.

[13] L. Cherkasova and G. Ciardo. Role of aging, frequency, and size in web cache replacement policies. In *Conf. on High-Performance Computing and Net.*, 2001.

[14] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *NSDI*, 2016.

[15] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SOCC*, 2010.

[16] F. J. Corbato. A paging experiment with the multics system. In Feshbach and Ingard, editors, *In Honor of Philip M. Morse*, pages 217–228. MIT Press, 1969.

[17] H. A. David and H. N. Nagaraja. *Order Statistics*. Wiley Series in Probability and Statistics, 2003.

[18] Django. `https://www.djangoproject.com`.

[19] G. Einziger and R. Friedman. TinyLFU: A highly efficient cache admission policy. In *Proc. Parallel, Dist. and Net. Processing*, 2014.

[20] J. B. Estoup. Gammes stenographiques., 1916.

[21] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *NSDI*, 2013.

[22] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, 1991.

[23] B. Fitzpatrick. Distributed caching with Memcached. *Linux J.*, 2004(124):5–5, 2004.

[24] P. A. Franaszek and T. J. Wagner. Some distribution-free aspects of paging algorithm performance. *J. ACM*, 21(1):31–39, 1974.

[25] S. Goel, A. Anderson, J. M. Hofman, and D. J. Watts. The structural virality of online diffusion. *Management Science*, 62(1):180–196, 2016.

[26] Guava: Google Core Libraries for Java. `https://github.com/google/guava`.

[27] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of Facebook photo caching. In *SOSP*, 2013.

[28] Hyperbolic caching. `https://github.com/kantai/hyperbolic-caching`.

[29] S. Jiang and X. Zhang. LIRS: an efficient low interreference recency set replacement policy to improve buffer cache performance. In *SIGMETRICS*, 2002.

[30] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *VLDB*, 1994.

[31] H. Kaplan, R. E. Tarjan, and K. Tsioutsiouliklis. Faster kinetic heaps and their use in broadcast scheduling. In *SODA*, pages 836–844, 2001.

[32] A. R. Karlin, S. J. Phillips, and P. Raghavan. Markov paging. *SIAM J. Computing*, 30(3):906–922, 2000.

[33] E. Koutsoupias and C. H. Papadimitriou. Beyond competitive analysis. *SIAM J. Computing*, 30(1):300–317, 2000.

[34] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C.-S. Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. Comput.*, 50(12):1352–1361, 2001.

[35] C. Li and A. L. Cox. GD-Wheel: A cost-aware replacement policy for key-value stores. In *EUROSYS*, 2015.

[36] C. Li, P. Shilane, F. Douglis, and G. Wallace. Pannier: A container-based flash cache for compound objects. In *Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, 2015.

[37] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *FAST*, 2003.

[38] Memcachier. `http://www.memcachier.com`.

[39] Multiworld Testing Decision Service. `http://aka.ms/mwt`.

[40] 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 V. Venkataramani. Scaling memcache at facebook. In *NSDI*, 2013.

[41] E. J. O'neil, P. E. O'neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. *SIGMOD Record*, 22(2):297–306, 1993.

[42] K. Psounis and B. Prabhakar. Efficient randomized web-cache replacement schemes using samples from past eviction times. *IEEE/ACM Trans. Networking*, 10(4):441–455, 2002.

[43] Q. Pu, H. Li, M. Zaharia, A. Ghodsi, and I. Stoica. Fairride: Near-optimal, fair cache sharing. In *NSDI*, pages 393–406, 2016.

[44] Redis Key-Value Store. `http://http://redis.io`.

[45] Using Redis as an LRU Cache. `https://redis.io/topics/lru-cache#approximated-lru-algorithm`.

[46] J. Shim, P. Scheuermann, and R. Vingralek. Proxy cache algorithms: Design, implementation, and performance. *IEEE Trans. Knowledge and Data Eng.*, 11(4):549–562, 1999.

[47] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Comm. ACM*, 28(2):202–208, 1985.

[48] Storage Performance Council Trace Repository. `http://www.storageperformance.org/specs/#traces`, 2002.

[49] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. RIPQ: Advanced photo caching on flash for Facebook. In *FAST*, 2015.

[50] Varnish HTTP Cache. `https://www.varnish-cache.org`.

[51] ViralSearch: Identifying and Visualizing Viral Content. `https://www.microsoft.com/en-us/research/video/viralsearch-identifying-and-visualizing-viral-content/`.

[52] Q. Yang and H. Zhang. Taylor series prediction: A cache replacement policy based on second-order trend analysis. In *HICSS*. IEEE, 2001.

[53] N. Young. *Competitive paging and dual-guided on-line weighted caching and matching algorithms*. PhD thesis, Princeton University, 1991.

[54] N. E. Young. On-line file caching. *Algorithmica*, 33(3):371–383, 2002.

[55] Y. Zhou, J. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *USENIX ATC*, 2001.

[56] G. K. Zipf. *Selected studies of the Principle of Relative Frequency in Language.* Harvard Univ. Press, 1932.

Execution Templates: Caching Control Plane Decisions for Strong Scaling of Data Analytics

Omid Mashayekhi Hang Qu Chinmayee Shah Philip Levis
Stanford University

Abstract

Control planes of cloud frameworks trade off between scheduling granularity and performance. Centralized systems schedule at task granularity, but only schedule a few thousand tasks per second. Distributed systems schedule hundreds of thousands of tasks per second but changing the schedule is costly.

We present *execution templates*, a control plane abstraction that can schedule hundreds of thousands of tasks per second while supporting fine-grained, per-task scheduling decisions. Execution templates leverage a program's repetitive control flow to cache blocks of frequently-executed tasks. Executing a task in a template requires sending a single message. Large-scale scheduling changes install new templates, while small changes apply edits to existing templates.

Evaluations of execution templates in Nimbus, a data analytics framework, find that they provide the fine-grained scheduling flexibility of centralized control planes while matching the strong scaling of distributed ones. Execution templates support complex, real-world applications, such as a fluid simulation with a triply nested loop and data dependent branches.

1 Introduction

As data analytics have transitioned from file I/O [1, 9] to in-memory processing [26, 28, 42], systems have focused on optimizing CPU performance [30]. Spark 2.0, for example, reports 10x speedups over prior versions with new code generation layers [38]. Introducing data-parallel optimizations such as vectorization, branch flattening, and prediction can in some cases be faster than hand-written C [32, 41]. GPU-based computations [2, 3] improve performance further.

Computational speedups, however, demand a higher task throughput from the control plane. This creates a tension between task throughput and dynamic, fine-grained scheduling. Available systems cannot fulfill both requirements simultaneously. Today, frameworks adopt one of two design points to schedule their computations across workers. One is a centralized controller model, and the other is a distributed data flow model.

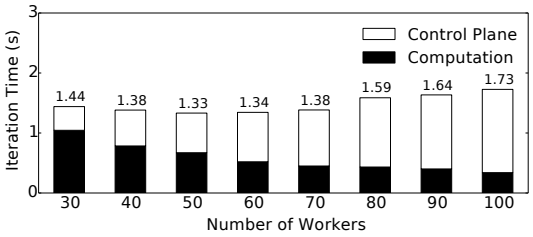


Figure 1: The control plane is a bottleneck in modern analytics workloads. Increasingly parallelizing logistic regression on 100GB of data with Spark 2.0's MLlib reduces computation time (black bars) but control overhead outstrip these gains, *increasing* completion time.

In the first model, systems such as Spark [42] use a centralized control plane, with a single node that dispatches small computations to worker nodes. Centralization allows a framework to quickly reschedule, respond to faults, and mitigate stragglers *reactively*, but as CPU performance improves the control plane becomes a bottleneck. Figure 1 shows the performance of Spark 2.0's MLlib logistic regression running on 30–100 workers. While computation time decreases with more workers, these improvements do not reduce overall completion time. Spark spends more time in the control plane, spawning and scheduling computations. While there is a huge body of work for scheduling *multiple* jobs within a cluster [6, 10, 11, 19, 23, 31, 35], these approaches do not help when a *single* job has a higher task throughput than what the control plane can handle, as in Figure 1.

The second approach, used by systems such as Naiad [28] and TensorFlow [3], is to use a fully distributed control plane. When a job starts, these systems install data flow graphs on each node, which then independently

execute and exchange data. By distributing the control plane and turning it into data flow, these frameworks achieve strong scalability at hundreds of thousands of tasks per second. However, data flow graphs describe a static schedule. Even small changes, such as rescheduling a task between two nodes, requires stopping the job, recompiling the flow graph and reinstalling it on every node. As a result, in practice, these systems mitigate stragglers only *proactively* by launching backup workers, which requires extra resource allocation even for non-straggling tasks [3].

This paper presents a new point in the design space, an abstraction called *execution templates*. Execution templates schedule at the same per-task granularity as centralized schedulers. They do so while imposing the same minimal control overhead as distributed execution plans.

Execution templates leverage the fact that long-running jobs (e.g. machine learning, graph processing) are repetitive, running the same computation many times [37]. Logically, a framework using execution templates centrally schedules at task granularity. As it generates and schedules tasks, however, the system caches its decisions and state in templates. The next time the job reaches the same point in the program, the system executes from the template rather than resend all of the tasks. Depending on how much system state has changed since the template was installed, a controller can immediately *instantiate* the template (i.e. execute without modification), *edit* the template by changing some of its tasks, or *install* a new version of template. Templates are not bound to a static control flow and support data-dependent branches; controllers *patch* system state dynamically at runtime if needed. We call this abstraction a template because it caches some information (e.g., dependencies) but instantiation requires parameters (e.g., task IDs).

Using execution templates, a centralized controller can generate and schedule hundreds of thousands of low-latency tasks per second. We have implemented execution templates in Nimbus, an analytics framework designed to support high performance computations. This paper makes five contributions:

1. Execution templates, a control plane abstraction that schedules high task throughput jobs at task granularity (Section 2).
2. A definition of the requirements execution templates place on a control plane and the design of Nimbus, a framework that meets these requirements (Section 3).
3. Details on how execution templates are implemented in Nimbus, including program analyses to generate and install efficient templates, validation and patching templates to meet their preconditions, and dynamic edits for in-place template changes (Section 4).

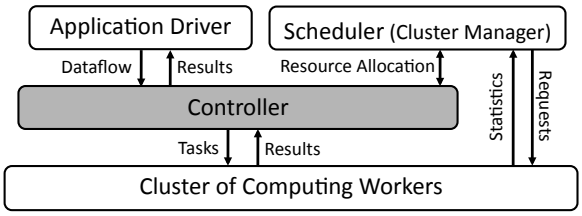


Figure 2: Generalized architecture of a cloud computing system: a *driver* program specifies the application logic to a *controller*, which can either directly assign tasks to workers or request resources from a *cluster manager*. Execution templates operate within a controller.

4. An evaluation of execution templates on analytics benchmarks, comparing them with Spark's fine-grained scheduler and Naiad's high-throughput data flow graphs (Section 5).
5. An evaluation of Nimbus running a PhysBAM [12] particle-levelset water simulation [13] with tasks as short as 100μs. (Section 5).[1]

This paper does not examine the question of scheduling policy, e.g., how to best place tasks on nodes, whether by min-cost flow computations [16, 21], packing [17, 18], or other algorithms [6, 19, 23, 33] (Section 6). Instead, it looks at the *mechanism*: how can a control plane support high throughput, fine-grained decisions? Section 7 discusses how execution templates can be integrated into existing systems and concludes.

2 Execution Templates

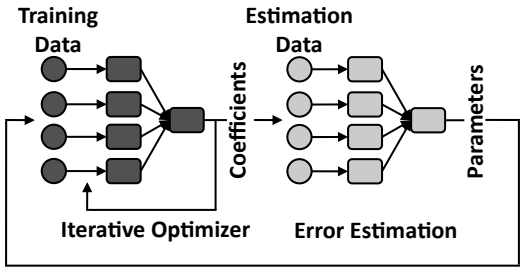
This section introduces execution templates and their characteristics. Figure 2 shows the general architecture of cloud computing systems. Execution templates operate on the controller and its interfaces.

Execution templates are motivated by the fact that long-running jobs are usually iterative and run same set of tasks repetitively [37] with minor changes. For example, Figure 3 shows the pseudocode and task graph for a training regression algorithm. The algorithm consists of a nested loop. The `Gradient` and `Estimate` operations can each generate many thousands of tasks. This graph structure is identical for each iteration, but the same vertex in two iterations can have different values across iterations, such as the `coeff` and `param` parameters. Furthermore, task identifiers change across iterations. With execution templates, the control plane can leverage the fixed structure to improve the performance.

[1] PhysBAM is an open-source simulation package that has received two Academy Awards and has been used in over 20 feature films.

```
while (error > threshold_e) {
   while (gradient > threshold_g) {
      // Optimization code block
      gradient = Gradient(tdata, coeff, param)
      coeff += gradient
   }
   // Estimation code block
   error = Estimate(edata, coeff, param)
   param = update_model(param, error)
}
```

(a) Driver program pseudocode.



(b) Iterative execution graph.

Figure 3: Task graph and driver program pseudocode of a training regression algorithm. It is iterative, with an outer loop for updating model parameters based on the estimation error, and an inner loop for optimizing the feature coefficients. The driver program has two basic blocks corresponding to inner and outer loops. `Gradient` and `Estimate` are both parallel operations that execute many tasks on partitions of data.

2.1 Abstraction

An execution template is a parameterizable list of tasks. The fixed structure of the template includes the list of tasks, their executable functions, task dependencies, relative ordering, and data access references. The parameter list includes the task identifiers and runtime parameters passed to each task.

To enable data dependent branches and nested loop structures, execution templates work at the granularity of basic blocks. A *basic block* is a code sequence in the driver program with only one entry point and no branches except the exit. For example, Figure 3 has two basic blocks, one for the inner loop and one for the outer loop operations. Note that loop unrolling and other batching techniques (e.g., as used in Drizzle [36]) cannot capture nested loops and data dependent branches.

Execution templates are *installed* and *instantiated* at run time. These two operations results in performance improvements in the control plane by caching and reusing repetitive control flow. Execution templates also support two special operations, *edits* and *patching*, which deal with scheduling changes and dynamic control flow. Each operation is discussed in the following subsections.

2.2 Installation and Instantiation

There are two types of execution templates, one for the driver-controller interface called a *controller template*, and one for the controller-worker interface called a *worker template*. Controller templates contain the complete list of tasks in a basic block across all of the worker nodes. They cache the results of creating tasks, dependency analysis, data lineage, bookkeeping for fault recovery, and assigning data partitions as task arguments. For every unique basic block, a driver program installs a controller template at the controller. The driver can then execute the same basic block again by telling the controller to instantiate the template.

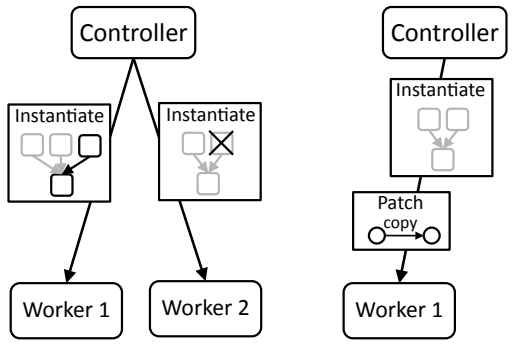
Where controller templates describes a basic block over the whole system, each worker template describes the portion of the basic block that runs on a particular worker. Workers cache the dependency information needed to execute the tasks and schedule them in the right order. Like TensorFlow [3], external dependencies such as data exchanges, reductions, or shuffles appear as tasks that complete when all data is transferred. Worker templates include metadata identifying where needed data objects in the system reside, so workers can directly exchange data and execute blocks of tasks without expensive controller lookups.

When a driver program instantiates a controller template, the controller makes a copy of the template and fills in all of the passed parameters. It then checks whether the prior assignment of tasks to workers matches existing worker templates. If so, it instantiates those templates on workers, passing the needed parameters. If the assignment has changed (e.g., due to scheduling away from a straggler or a failure), it either edits worker templates or installs new ones. In the steady state, when two iterations of a basic block run on the same set of n workers, the control plane sends $n+1$ messages: one from the driver to the controller and 1 from the controller to each of the n workers.

2.3 Edits

Execution templates have two mechanisms to make control plane overhead scale gracefully with the size of scheduling changes: installing new templates and editing existing ones. If the controller makes large changes to a worker's tasks, it can install a new worker template. Workers cache multiple worker templates, so a controller can move between several different schedules by invoking different sets of worker templates.

Edits allow a controller to change an existing worker template. Figure 4(a) shows how edits manifest in the



(a) Edit task from worker 2 to 1. (b) Patching.

Figure 4: Edits and patches allow a framework to efficiently adapt templates to dynamic changes in the system. Edits dynamically modify a template structure in place, while patches move and copy data objects to match a template's preconditions. Grey denotes cached template information, while black denotes information sent over the network.

control plane: they modify already installed templates in place. Edits are used when the controller needs to make small changes to the schedule, e.g., migrate one of many partitions. Edits are included as metadata in a worker template instantiation message and modify its data structures. An edit can remove and add tasks. Edits keep the cost of dynamic scheduling proportional to the extent of changes. Together, installation and edits allow a controller to make fine-grained changes to how a basic block is distributed across workers.

2.4 Patching

Each worker template has a set of preconditions that must hold when the template is instantiated, for example requiring a replicated data object in local memory to have the most recent write. When a driver program instantiates a controller template, the system state may not meet the preconditions of the associated worker templates. This can happen because a basic block can be entered from many different points in the program. When a template is created, the controller may not even have seen all of these positions (e.g., an edge case covered by an if/else statement).

A controller uses patches to ensure correct execution in the presence of dynamic driver program control flow. Patches update and move data from one worker to another to satisfy the preconditions. For example, the worker templates for the inner loop in Figure 3(a) have the precondition that `param` needs to be in local memory. But there are two cases in which the controller might invoke the templates: the first iteration of the loop and subsequent iterations. In subsequent iterations, `param` is inductively already in local memory. However, on the first iteration, `param` exists only on the worker that calculated it. The controller therefore *patches* the inner loop template, sending directives to workers that copy `param` to each worker (Figure 4(b)).

Patches allow execution templates to efficiently handle dynamic program control flow. This is important when loop conditions are based on data, such as running until an error value falls below a threshold. There are two options to deal with the associated uncertainties in control flow. The controller can either ensure that the preconditions of every worker template always hold, or when a template is instantiated it can patch system state to match the preconditions. The first approach is prohibitively expensive, because it requires unnecessary and expensive data copies. For example, it would require immediately copying `param` in Figure 3(a) to every worker after it is calculated even if the outer loop terminates. A controller therefore has to react to the driver's stream of controller template instantiation requests and enforce the preconditions on the fly.

3 System Design

This section defines the requirements that execution templates place on a control plane and describes the design of a cloud computing framework, called Nimbus, that meets these requirements.

3.1 Control Plane Requirements

Conceptually, execution templates can be incorporated into any existing cloud framework. Incorporating them, however, assumes certain properties in the framework's control plane. We describe these requirements here, and defer a discussion of how they can be incorporated into existing systems to Section 7.

1. Workers maintain a queue of tasks and locally determine when tasks are runnable. Worker templates create many tasks on a worker, most of which are not immediately runnable because they depend on the output of prior tasks. A worker must be able to determine when these tasks are runnable without going through a central controller, which would become a bottleneck.

2. Workers can directly exchange data. Within a single template, one worker's output can be the input of tasks on other workers. As part of executing the template, the two workers need to exchange data without going through a central controller, which would become a bottleneck.

3. The controller schedules fine-grained tasks. Fine-grained tasks are a prerequisite to support fine-grained scheduling; they define the minimum scheduling change that a system can support.

3.2 Nimbus Architecture

Nimbus is an analytics framework that meets the three requirements. Nimbus's system architecture is designed to support execution templates and run computationally intensive jobs that operate on in-memory data across many nodes. Like Spark, Nimbus has a centralized controller node that receives tasks from a driver program. The controller dispatches these application tasks to workers. The controller is responsible for transforming tasks from a driver program into an execution plan, deciding on which workers to run which computations.

As it sends application tasks to workers, the controller inserts additional control tasks, such as tasks to copy data from one worker to another. These tasks explicitly name the workers involved in the transfer, such that workers can directly exchange data.

3.3 Nimbus Execution and Data Model

In Nimbus, a job is decomposed into *stages*. A stage is a computation over a set of input data and produces a set of output data. Each data set is partitioned into many *data objects* so that stages can be parallelized. Each stage typically executes many *tasks*, one per object, that operate in parallel. In addition to the identifiers specifying the data objects it accesses, each task can be passed parameters, such as model parameters or constants.

Nimbus tasks operate on mutable data objects. Supporting in-place modification of data avoids data copies and are crucial for computational and memory efficiency. In-place modification also has two crucial benefits for execution templates. First, multiple iterations of a loop access the same objects and reuse their identifiers. This means the data object identifiers can be cached in a template, rather than recomputed on each iteration. Second, mutable data objects reduce the overall number of objects in the system by a large constant factor, which improves lookup speeds.

Mutable objects mean there can be multiple copies and versions of an object in the system. For example, for the code in Figure 3(a), after the execution of the outer loop, there are n copies of `param`, one on each worker. However, one copy of `param`, has been written to, and has an updated value. Each data object in the system therefore combines an object identifier with a version number. The Nimbus controller ensures, through data copies, that tasks on a worker always read the latest value according to the program's control flow.

3.4 Nimbus Control Plane

The Nimbus control plane has four major commands. Data commands create and destroy data objects on workers. Copy commands copy data from one data object to another (either locally or over a network). File commands load and save data objects from durable storage. Finally, task commands tell the worker to execute an application function.

Commands have five fields: a unique identifier, a *read set* of data objects to read, a *write set* of data objects to write, a *before set* of the commands that must complete before this one can execute, and a binary blob of *parameters*. Task commands include a sixth field, which application function to execute.

A command's before set includes only other tasks on that worker. If there is a dependency on a remote command, this is encoded through a copy command. For example a task associated with the `update_model` operation in Figure 3(a) depends on the results of the parallel `Estimate` operation. The `update_model` task has n copy commands in its before set; one for each locally computed `error` in each partition.

Copy commands execute asynchronously and follow a push model. A sender starts transmitting an object as soon as the command's before set is satisfied. Because this uses asynchronous I/O it does not block a worker thread. Similarly, a worker asynchronously reads data into buffers as soon as it arrives. Once the before set of a task reading the data is satisfied, worker changes a pointer in the data object to point to the new buffer.

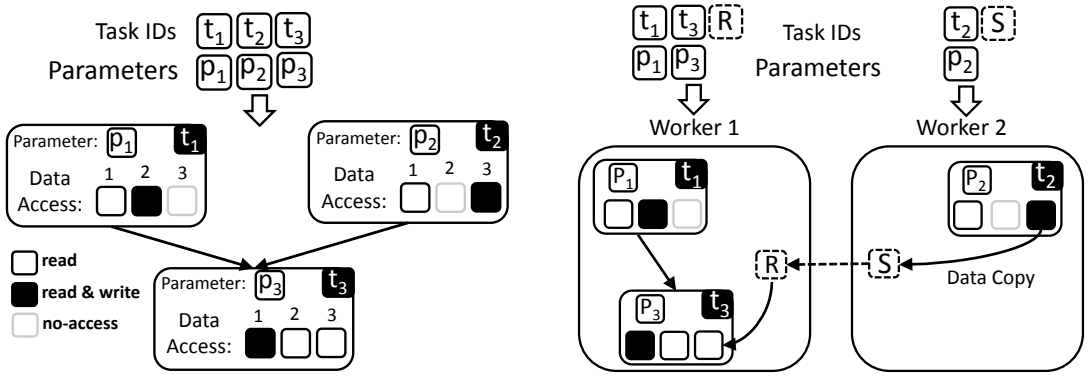
4 Implementation

This section describes how Nimbus implements execution templates and their operations. The Nimbus code, including execution templates, is publicly available at `https://github.com/omidm/nimbus`. Nimbus core library is about 35,000 semicolons of C++ code and supports tasks written in C++. In addition to machine learning and graph processing applications, the repository includes graphical simulations ported to Nimbus.

4.1 Installation and Instantiation

Template installation begins with the driver sending a start template message to the controller at the beginning of a basic block. In the current implementation, a programmer explicitly marks the basic block in the driver program. For example, in Figure 3(a), the only change in the driver program to support templates is extra annotations before and after basic blocks; one can also use other automatic approaches such as static program analysis. As the controller receives tasks, it simultaneously schedules them normally and stores them in a temporary task graph data structure.

At the end of the basic block, the driver sends a template finish message. On receiving a finish message, the controller takes the task graph and post-processes it into



(a) A **controller template** represents the common structure of a task graph metadata. It stores task dependencies and data access patterns. It is invoked by filling in task identifiers and parameters to each task.

(b) Each **worker template** stores the common structure of a task graph for execution including the data copies among workers. It is invoked by passing the task identifiers, and parameters to each task.

Figure 5: Controller template and worker templates for a simple task graph.

an optimized, table-based data structure. Pointers are turned into indexes for fast lookups into arrays of values.

Controller templates cache the read set, write set, and function identifier. A template instantiation message includes an array of command identifiers and a block of task parameters. Within a template, task identifiers index into this array. The one time cost of generating the ordered indices keeps the successive instantiations efficient. Figure 5(a) shows the instantiation of a controller template with new set of task identifiers and parameters.

Once it has generated the controller template, the controller generates the associated worker templates. Worker templates have two halves. The first half exists at the controller and represents the entire execution across all of the workers. This centralized half allows the controller to cache how the template's tasks are distributed across workers and track the preconditions for generating patches when needed.

Each worker template has preconditions that list which data objects at each worker must hold the latest update to that object. Not all data objects are required to be up to date: a data object might be used for writing intermediate data and be updated within the worker template itself. For example, in Figure 5(b), the third data object on worker 1 does not need to have the latest update at the beginning of the worker template; the data copy within the worker template updates it.

The second half of the worker template is distributed across the workers and caches the per-worker local command graph which they locally schedule. The controller installs worker templates very similarly to how the driver installs controller templates. And like controller templates, instantiation passes an array of task identifiers and parameters. Figure 5(b) shows a set of worker templates for controller template in Figure 5(a).

4.2 Patching

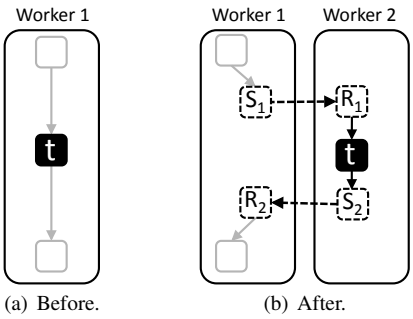
Before instantiating a worker template, controller must *validate* whether the template's preconditions hold and patch the worker's state if not. Validating and patching must be fast, because they are sequential control plane overhead that cannot be parallelized. Making them fast is challenging, however, when there are many workers, data objects, and tasks, because they require checking a great deal of state.

Nimbus uses two optimizations to keep validation and patching fast. The first optimization relates to template generation. When generating a worker template, Nimbus ensures that the precondition of the template holds when it finishes. By doing so, it ensures that tight inner loops, which dominate execution time and control plane traffic, automatically validate and need no patching. As an example, in Figure 5(b), this adds a data copy of object 1 to worker 2 at the end of the template.

Second, workers cache patches and the controller can invoke these patches much like a template. When a worker template fails validation, the controller checks a lookup table of prior patches indexed by what executed before that template. If the cached patch will correctly patch the template, it sends a single command to the worker to instantiate the patch. When patches require multiple data copies, the cache helps reduce the networking overhead at the controller. We have found that the patch cache has a very high hit rate in practice because control flow, while dynamic, is typically not very complex.

4.3 Edits

Whenever a controller instantiates a worker template, it can attach a list of edits for that template to apply before



(a) Before. (b) After.

Figure 6: Edits to reschedule a task. The controller removes the task from worker 1's template and adds two data copy commands (S_1, R_2). It adds the task and two data copy commands (R_1, S_2) to worker 2's template.

instantiation. Each edit specifies either a new task to include or a task to remove. Edits are usually limited to the actual tasks being added or removed, because in cases when there are dependencies with other tasks, tasks are exchanged with data copy commands. Figure 6 shows, for example, how a task's entry in a before set is replaced by a data receive command. As long as the data receive command is assigned the same index within the command identifier array, other commands do not need to change. Using edits, minor changes in scheduling have very small costs. The cost scales linearly with the size of the change.

4.4 Fault Recovery

Nimbus implements a checkpoint recovery mechanism. Although a controller keeps the full lineage for every data object in the system, for iterative computations we found that linage-based recovery [42] is essentially identical to checkpointing because there are frequent synchronization points around shared global values. Any lineage recovery beyond a synchronization point requires regeneration of every data object, which is a checkpoint.

Nimbus automatically inserts checkpoints into the task stream from a driver program. When a checkpoint triggers, the controller waits until all worker task queues drain, stores a snapshot of the current execution graph, and requests every worker to write its live data objects to durable storage.

When a controller determines a worker has failed (it stops sending periodic heartbeat messages or workers depending on its data fall idle), it sends a halt command to every worker. On receiving the command, workers terminate all ongoing tasks, flush their queues, and respond back. Then, the controller sends commands to load the latest checkpoint into memory, reverts to the stored execution graph snapshot, and restarts execution.

5 Evaluation

This section evaluates execution templates in Nimbus, comparing them with Spark's fine-grained centralized scheduler, Naiad's high-throughput distributed data flow graphs [2], and application-level MPI messaging. In summary, the results show:

- Execution templates allow Nimbus to schedule hundreds of thousands of tasks per second, imposing a control overhead competitive with Naiad's distributed data flow graphs.
- Execution templates allow Nimbus to schedule at task granularity, providing a runtime flexibility and adaptivity equivalent to Spark's centralized scheduler.
- Execution templates are expressive enough to support complex, high-performance applications, such as a particle-levelset water simulation with a triply nested, data dependent loop and tasks as short as 100μs.

5.1 Methodology

All experiments use Amazon EC2 compute-optimized instances since they are the cheapest option for compute-bound workloads. Worker nodes use `c3.2xlarge` instances with 8 virtual cores and 15GB of RAM. Controllers run on a more powerful `c3.4xlarge` instance to show how jobs bottleneck on the controller even when it has more resources. All nodes are allocated in a single placement group and so have full bisection bandwidth.

We compare the performance of Nimbus with Spark 2.0 and Naiad 0.4.2 using two machine learning benchmarks, logistic regression and k-means clustering. Because our goal is to measure the task throughput and scheduling granularity of the control plane, we factor out language differences between the three frameworks and have them run tasks of equal duration. We chose the task duration as the fastest of the three frameworks, as it evaluates the highest task throughput. Nimbus tasks run 8 times faster than Spark's MLlib due to Spark using a JVM (a 4x slowdown) and its immutable data requiring copies (a 2x slowdown). Nimbus tasks run 3 times faster than Naiad due to Naiad's use of the CLR. To show that tasks in Naiad and Spark run as fast as C++ ones, we label them *Naiad-opt* and *Spark-opt*. This is done by replacing the task computations with a spin wait as long as C++ tasks.

The Naiad and Nimbus implementations of k-means and logistic regression include application-level two-level reduction trees. Application-level reductions in Spark harm completion time because they add more tasks that bottleneck at the controller.

[2] TensorFlow's control plane design is very similar to Naiad's which results in very close performance and behavior.

	Per-task cost
Installing controller template	25μs
Installing worker template on controller	15μs
Installing worker template on worker	9μs
Nimbus task scheduling	134μs
Spark task scheduling	166μs

Table 1: Template installation is fast compared to scheduling. The 49μs per-task cost is evenly split between the controller and worker templates. Installing a new worker template has a per-task cost of 24μs, and 18% overhead on centrally scheduling that task.

	Per-task cost
Instantiate controller template	0.2μs
Instantiate worker template	
with auto-validation	1.7μs
with explicit-validation	7.3μs

Table 2: Template instantiation is fast. For the common case of a template automatically validating (repeated execution of a loop), instantiation takes 1.9μs/task: Nimbus can schedule over 500,0000 tasks/sec. If dynamic control flow requires a full validation, it takes 7.5μs/task and Nimbus can schedule 130,000 tasks/second.

5.2 Micro-Benchmarks

This section presents micro-benchmark performance results. These results are from a logistic regression job with a single controller template with 8,000 tasks, split into 100 worker templates with 80 tasks each.

Table 1 shows the costs of template installation. We report the per-task costs because they scale with the number of tasks (there are individual task messages). We also report the cost of centrally scheduling a task in Spark and Nimbus to give context. Installing a template has a one-time cost of installing the controller template and the potentially repeated cost of installing worker templates. Adding a task to a controller template takes 25μs. Adding it to a worker template takes 24μs. In comparison to scheduling a task (134μs), this cost is small. Installing all templates has an overhead of 36% on centrally scheduling tasks.

Table 2 shows the costs of template instantiation. There are two cases for the worker template. In the first (common) case, the template validates automatically because it is instantiated after the same template. Since Nimbus ensures that a template, on completion, meets its preconditions, in this case the controller can skip validation. In the second case, a different worker template is instantiated after the previous one, and controller must fully validate the template. When executing the inner loop of a computation, Nimbus's scheduling throughput is over 500,000 tasks/second (0.2μs + 1.7μs per task).

	Cost
Nimbus single edit	$\approx$ 41μs
Nimbus rescheduling 5% of tasks (800 edits)	35ms
Nimbus complete installation (8000 tasks)	203ms
Naiad any scheduling change	230ms

Table 3: A single edit to the logistic regression job takes 41μs in Nimbus, and the cost scales linearly with the number of edits. Edits are less expensive than full installation when rescheduling as high as 5% of the tasks in templates. Any change in Naiad induces the full cost of data flow installation.

Table 3 shows the costs of edits. A single edit (removing or adding a task) takes 41μs[3]. Edits allow controllers to inexpensively make small-scale changes to worker templates. For example, 800 edits (e.g., rescheduling 5% of the tasks) takes 35ms, fraction of complete installation cost. The cost of installing physical graphs on Naiad, caused by any change to the schedule, is about 230ms.

5.3 Control Plane Performance

This section evaluates the scalability of execution templates and their impact on job completion time. Figure 7 shows the results of running logistic regression and k-means clustering over a 100GB input once data has been loaded and templates have been installed. We observed negligible variance in iteration times and report the average of 30 iterations.

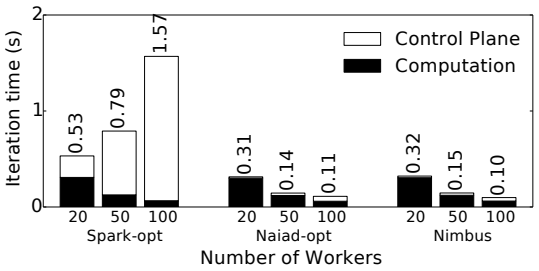
Nimbus and Naiad have equivalent performance. With 20 workers, an iteration of logistic regression takes 210-220ms; with 100 workers it takes 60-80ms. The slightly longer time for Naiad with 100 workers (80ms) is due to the Naiad runtime issuing many callbacks for the small data partitions; this is a minor performance issue and can be ignored. For k-means clustering, an iteration across 20 nodes takes 310-320ms and an iteration across 100 nodes takes 100-110ms. Completion time shrinks slower than the rate of increased parallelism because reductions do not parallelize.

Running over 20 workers, Spark's completion time is 70-100% longer than Nimbus and Naiad. With greater parallelism (more workers), the performance difference increases: Naiad and Nimbus run proportionally faster and Spark runs slower. Over 100 workers, Spark's completion time is 15-23 times longer than Nimbus. The dif-

[3]It is greater than the cost of installing a task in a worker template (29μs) due to the necessary changes in the task graph and inserting extra copy tasks (see Figure 6).



(a) Logistic regression



(b) K-means clustering

Figure 7: Iteration time of logistic regression and k-means for a data set of size 100GB. Nimbus executes tasks implemented in C++. Spark-opt and Naiad-opt show the performance when the computations are replaced with spin-wait as fast as tasks in C++. Execution templates helps centralized controller of Nimbus scale out almost linearly.



Figure 8: Task throughput of Nimbus and Spark as the number of workers increases. Spark saturates at about 6,000 tasks per second, while Nimbus grows to adapt to the number of tasks required for more parallelism. Note that the y-axis scale is different in the plots.

ference is entirely due to the control plane. Spark workers spend most of the time idle, waiting for the Spark controller to send them tasks.

Figure 8 shows the rate at which Nimbus and Spark schedule logistic regression tasks as the number of workers increases. Spark quickly bottlenecks at 6,000 tasks per second. Nimbus scales to support the increasing task throughput: a single iteration over 100 workers takes 60ms and executes 8,000 tasks, which is 128,000 tasks/second (25% of Nimbus's maximum throughput). Note that greater parallelism increases the task rate superlinearly because it simultaneously creates more tasks and makes those tasks shorter.

5.4 Dynamic Scheduling

Figure 9 shows the scenario of running a logistic regression job over 100 workers and rescheduling 5% of tasks every 5 iterations. The incremental edit cost lets Nimbus finish 20 iterations almost twice as fast as Naiad. Even if there is a single task rescheduling, Naiad would

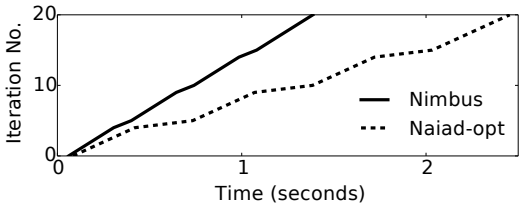


Figure 9: Logistic regression over 100 workers with task rescheduling every 5 iterations. Nimbus's edits have negligible overhead, while Naiad requires complete data flow installation for any scheduling change.

behave similarly; however, Nimbus's overhead remains negligible even in such an extreme scenario. Note that, current Naiad implementation does not support any data flow flexibility once the job starts, so the curve here is simulated from the numbers in Table 3 and Figure 7(a).

Figure 10 shows the time per iteration of logistic regression in Nimbus as a cluster manager adjusts the available resources. The run starts with templates disabled: the control plane overhead of a centralized scheduler dominates iteration time: each iteration takes 1.07s. At iteration 10, the driver starts using templates. Iteration 10 takes $\approx$ 1.3s, as installing each of the 8,000 tasks in the controller template adds 25μs (Table 2). On iteration 11, the controller template has been installed, and the controller generates its half of the worker template as it continues to send individual tasks to workers. This iteration is faster because the control traffic between the driver and controller is a single instantiation message. On iteration 12, the controller half of the worker templates has been installed, and the controller sends tasks to and installs templates on the workers. On iteration 13, templates are fully installed and an iteration takes 60ms (as in Figure 7(a)), with minimal control plane overhead.

At iteration 20, the cluster resource manager revokes 50 workers from the job's allocation. On this iteration, the controller regenerates the controller half of the

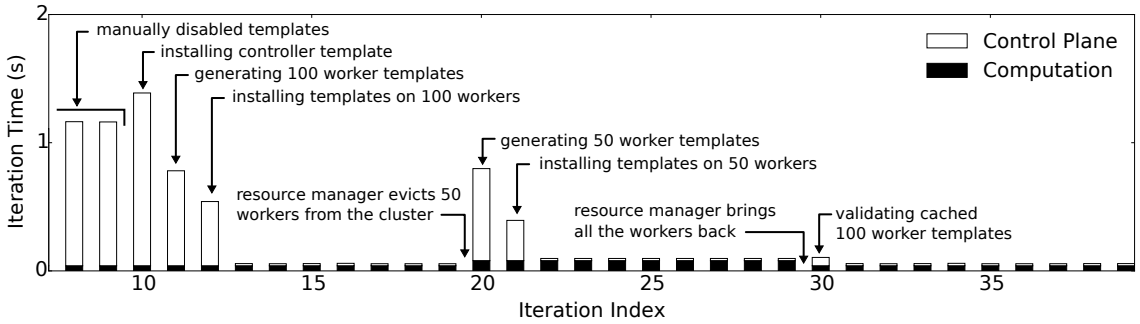
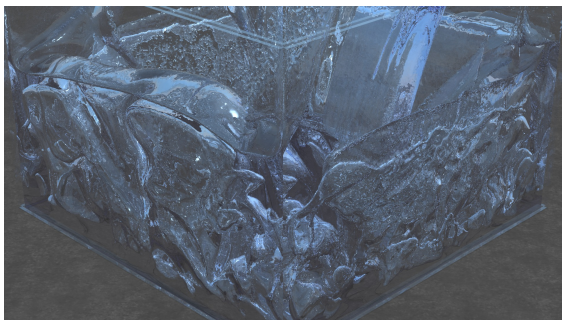


Figure 10: Execution templates can schedule jobs with high task throughputs while dynamically adapting as resources change. This experiment shows the control overheads as a cluster resource manager allocates 100 nodes to a job, revokes 50 of the nodes, then later returns them.



(a) Still of water pouring into a glass bowl.

(b) Iteration time of the main outer loop.

Figure 11: PhysBAM water simulation.

worker template, rescheduling tasks from evicted workers to remaining workers. On iteration 21, the controller installs new worker templates on the 50 workers. Computation time doubles because each worker is performing twice the work.

At iteration 30, the cluster resource manager restores the 50 workers to the job's allocation. The controller reverts to using the original worker templates and so does not need to install templates. However, on this first iteration, it needs to validate the templates. After this explicit validation, each iterations takes 60ms.

5.5 Complex Applications

To evaluate if execution templates can handle full applications with complex control flows, we use PhysBAM, an open-source computer graphics simulation library [12]. We ported PhysBAM to Nimbus, wrapping PhysBAM functions inside tasks and interfacing PhysBAM data objects (level sets, mark-and-cell grids, particles) into Nimbus.

We ran a canonical particle-levelset fluid simulation benchmark, water being poured into a glass [13]. This is the same core simulation used for the ocean in The Perfect Storm and the river in Brave. It has a triply-nested loop with 21 different computational stages that access over 40 different variables. The driver program has 8 basic blocks, three of them require extra data copies for auto validation, and two of them have non-deterministic entry points with different patch sets. Systems with static data flow (e.g., Naiad) cannot run this simulation efficiently because the termination conditions of its two inner loops are based on data values. Without data dependent branches, each loop instance must run as many iterations as the longest instance, which is wasteful when the loop converges faster.

We ran a 1024^3 cell simulation (512GB-1TB of RAM) on 64 workers. The median task length is 13ms, 10% of tasks are <3ms and some tasks are as short as 100μs. Figure 11 shows the results of running the simulation with PhysBAM's hand-tuned MPI libraries, in Nimbus without templates and in Nimbus with templates. The MPI libraries cannot rebalance load, and in practice developers rarely use them due to their brittle behavior and lack of fault tolerance. Without templates, the central controller becomes the bottleneck and the simulation takes 520% longer than MPI. With templates, the simulation runs within 15% of the MPI implementation, while providing fine-grained scheduling, automatic fault tolerance, and adaptive load balancing.

6 Related Work

Execution templates build on a large body of prior work that can be divided into three major categories: cloud

frameworks, cloud schedulers, and high performance computing.

Cloud frameworks schedule tasks from a single job. Systems such as CIEL [29], Spark [42] and Optimus [24] keep all execution state on a central controller, dynamically dispatching tasks as workers become ready. This gives the controller an accurate, global view of the job's progress, allowing it to quickly respond to failures, changes in available resources, and system performance. Execution templates borrow this model, but cache control plane decisions to drastically increase task throughput for strong scalability.

Systems such as Naiad [28] and TensorFlow [3] take the opposite approach, statically installing an execution plan on workers so the workers can locally generate tasks and directly exchange data. Execution templates borrow this idea of installing execution plans at runtime but generalize it to support multiple active plans and dynamic control flow. Furthermore, execution templates maintain fine-grained scheduling by allowing a controller to edit the current execution plan.

Frameworks such as Dryad [20], DryadLINQ [40], and FlumeJava [8], as well as programming models such as DimWitted [25], DMLL [7] and Spark optimizations [32, 41, 4, 39, 38] focus on abstractions for parallel computations that enable optimizations and high performance, in some cases faster than hand-written C. This paper examines a different but complementary question: how can a framework's runtime scale to support the resulting fast computations across many nodes?

Cloud schedulers (also called cluster managers) schedule tasks from many concurrent jobs across a collection of worker nodes. Because these schedulers have global knowledge of all of the tasks in the system, they can efficiently multiplex jobs across resources[17], improve job completion time [14], fairly allocate resources across jobs [15], follow other policies [6, 11, 19], or allow multiple algorithms to operate on shared state [33].

Traditional centralized schedulers have transitioned to distributed or hybrid models. In Sparrow [31], each job runs its own independent scheduler that monitors the load on workers. These schedulers independently make good cooperative scheduling decisions based on mechanisms and principles derived from the power of two choices [27]. Tarcil uses a coarser grained approach, in which multiple schedulers maintain copies of the full cluster state, whose access is kept efficient through optimistic concurrency control because conflicts are rare [11]. Hawk's hybrid approach centrally schedules long-running jobs for efficiency and distributes short job scheduling for low latency [10]. Mercury allows multiple schedulers to request resources from a shared pool and then schedule tasks on their resources [23].

These distributed and hybrid schedulers address the problem of when the combined task rate of multiple jobs is greater than what a centralized scheduler can handle. Execution templates solve a similar, but different problem, when the control plane bottlenecks a *single* job. Like Sparrow, a framework using execution templates requests allocation from its cluster manager.

High performance computing (HPC) embraces the idea that an application should be responsible for its own scheduling as it has the greatest knowledge about its own performance and behavior. HPC systems stretch from very low-level interfaces, such as MPI [34], which is effectively a high performance messaging layer with some support for common operations such as reduction. Partitioning and scheduling, however, is completely an application decision, and MPI provides very little support for load balancing or fault recovery. HPC frameworks such as Charm++ [22] and Legion [5] provide powerful abstractions to decouple control flow, computation and communication, similar to cloud frameworks. Their fundamental difference, however, is that these HPC systems only provide mechanisms; applications are expected to provide their own policies.

7 Discussion and Conclusion

Analytics frameworks today provide either fine-grained scheduling or high task throughput but not both. Execution templates enable a framework to provide both simultaneously. By caching task graphs on the controller and workers, execution templates are able to schedule half a million tasks per second (Table 2). At the same time, controllers can cheaply edit templates in response to scheduling changes (Table 3). Finally, patches allow execution templates to support dynamic control flow.

Execution templates are a general control plane abstraction. However, the requirements listed in Section 3 are simpler to incorporate in some systems than others. Incorporating execution templates into Spark requires two changes: workers need to queue tasks and resolving dependencies locally and workers need to be able to exchange data directly (not go through the controller for lookups). Naiad's data flow graphs as well as TensorFlow's can be thought of as an extreme case of execution templates, in which the flow graph describes a very large, long-running basic block. Allowing a driver to store multiple graphs, edit them, and dynamically trigger them would bring most of the benefits.

Acknowledgments This work was funded by the National Science Foundation (CSR grant #1409847) and conducted in conjunction with the Intel Science and Technology Center - Visual Computing. The experiments were made possible by a generous grant from the Amazon Web Services Educate program.

References

[1] Apache Hadoop. http://wiki.apache.org/hadoop.

[2] Facebook AI Research open sources deep-learning modules for Torch. https://research.facebook.com/blog/fair-open-sources-deep-learning-modules-for-torch/.

[3] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

[4] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[5] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012.

[6] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 285–300, 2014.

[7] K. J. Brown, H. Lee, T. Rompf, A. K. Sujeeth, C. De Sa, C. Aberger, and K. Olukotun. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 194–205. ACM, 2016.

[8] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *ACM Sigplan Notices*, volume 45, pages 363–375. ACM, 2010.

[9] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[10] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. Hawk: hybrid datacenter scheduling. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 499–510, 2015.

[11] C. Delimitrou, D. Sanchez, and C. Kozyrakis. Tarcil: reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 97–110. ACM, 2015.

[12] P. Dubey, P. Hanrahan, R. Fedkiw, M. Lentine, and C. Schroeder. Physbam: Physically based simulation. In *ACM SIGGRAPH 2011 Courses*, SIGGRAPH '11, pages 10:1–10:22, New York, NY, USA, 2011. ACM.

[13] D. Enright, R. Fedkiw, J. Ferziger, and I. Mitchell. A hybrid particle level set method for improved interface capturing. *Journal of Computational Physics*, 183(1):83–116, 2002.

[14] M. R. Garey, D. S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of operations research*, 1(2):117–129, 1976.

[15] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, volume 11, pages 24–24, 2011.

[16] I. Gog, M. Schwarzkopf, A. Gleave, R. N. M. Watson, and S. Hand. Firmament: Fast, centralized cluster scheduling at scale. In *To appear in Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX, 2016.

[17] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 455–466. ACM, 2014.

[18] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. Do the hard stuff first: Scheduling dependent computations in data-analytics clusters. *arXiv preprint arXiv:1604.07371*, 2016.

[19] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

[21] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276. ACM, 2009.

[22] L. V. Kale and S. Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.

[23] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 485–497, 2015.

[24] Q. Ke, M. Isard, and Y. Yu. Optimus: a dynamic rewriting framework for data-parallel execution plans. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 15–28. ACM, 2013.

[25] J. Liu and S. J. Wright. Asynchronous stochastic coordinate descent: Parallelism and convergence properties. *SIAM Journal on Optimization*, 25(1):351–376, 2015.

[26] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[27] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[28] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[29] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: A universal execution engine for distributed data-flow computing. In *NSDI*, volume 11, pages 9–9, 2011.

[30] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B.-G. Chun, and V. ICSI. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 293–307, 2015.

[31] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[32] S. Palkar, J. Thomas, and M. Zaharia. Nested vector language: Roofline performance for data parallel code. http://livinglab.mit.edu/wp-content/uploads/2016/01/nvl-poster.pdf.

[33] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364. ACM, 2013.

[34] M. Snir. *MPI–the Complete Reference: The MPI core*, volume 1. MIT press, 1998.

[35] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[36] S. Venkataraman, A. Panda, K. Ousterhout, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale.

[37] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *Proceedings of the 13th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2016.

[38] R. Xin. Technical Preview of Apache Spark 2.0 Now on Databricks. https://databricks.com/blog/2016/05/11/apache-spark-2-0-technical-preview-easier-faster-and-smarter.html.

[39] R. Xin and J. Rosen. 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.

[40] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, volume 8, pages 1–14, 2008.

[41] M. Zaharia. New developments in spark and rethinking apis for big data. http://platformlab.stanford.edu/Seminar%20Talks/stanford-seminar.pdf.

[42] 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*. USENIX Association, 2012.

*c*Hash: Detection of Redundant Compilations via AST Hashing

Christian Dietrich*, Valentin Rothberg*, Ludwig Füracker+,
Andreas Ziegler+ and Daniel Lohmann*

+Friedrich-Alexander Universität Erlangen-Nürnberg
*Leibniz Universität Hannover

Abstract

Software projects that use a compiled language are built hundreds of thousands of times during their lifespan. Hence, the compiler is invoked over and over again on an incrementally changing source base. As previous work has shown, up to 97 percent of these invocations are redundant and do not lead to an altered compilation result. In order to avoid such redundant builds, many developers use caching tools that are based on textual hashing of the source files. However, these tools fail in the presence of modifications that leave the compilation result unchanged. Especially for C projects, where module-interface definitions are imported textually with the C preprocessor, modifications to header files lead to many redundant compilations.

In this paper, we present the *c*Hash approach and compiler extension to quickly detect modifications on the language level that will not lead to a changed compilation result. By calculating a hash over the abstract syntax tree, we achieve a high precision at comparatively low costs. While *c*Hash is light-weight and build system agnostic, it can cancel 80 percent of all compiler invocations early and reduce the build-time of incremental builds by up to 51 percent. In comparison to the state-of-the-art CCache tool, *c*Hash is at least 30 percent more precise in detecting redundant compilations.

1 Introduction

Software development for a project that uses a compiled language involves a (seemingly) endless number of compiler invocations. Typically, a developer edits some source files, builds the whole project, and then tests and debugs the resulting binary. In this process, which is repeated tens to hundreds of times a day by thousands of developers, the time taken for the (noninteractive) build step is a crucial property to developer productivity [23].

After many incremental modifications, software developers typically commit their changes into a larger project-wide repository. From there, the robots of a continuous integration platform might pull and merge them to perform automated build tests, which involves some additional thousand builds of the software. A prominent example is Linux and the Intel 0-day robot.[1] The robot monitors more than 600 development repositories to run tests on newly integrated changes, for which it builds more than 36 000 Linux kernels on an average day in order to provide kernel developers with quick feedback on integration issues. Again, the time of each build is a crucial property for the effectiveness of the system – the more builds it can handle each day, the more build tests can be performed.

1.1 Redundant Builds

In both settings, the build process itself can often be performed as an *incremental build*: Compilation is generally considered to be an idempotent operation. Hence, only the source modules that are affected by a change or commit – either directly or transitively via a dependency – need to be recompiled into object files, while a large portion of unchanged object files can be reused from a previous build. In larger projects, build times thereby are reduced from hours and minutes for a full build to seconds and milliseconds for an incremental build.

The challenge here is to detect – in a reliable but fast manner – which source modules are part of the increment that needs to be recompiled. Ideally, a module becomes part of the increment only if its recompilation would lead to a program with a different *behavior* – which is undecidable in the general sense. Therefore, we technically reduce this to the decision if recompilation would lead to a different program *binary*. The respective test needs to be reliable in that it never produces a false negative (i.e., excludes some source module from the increment that is affected by a change). False positives do not harm reliability, but lead to costly *redundant builds* – which we wanted

[1] https://lwn.net/Articles/514278/

```
objects = network.o main.o filesys.o

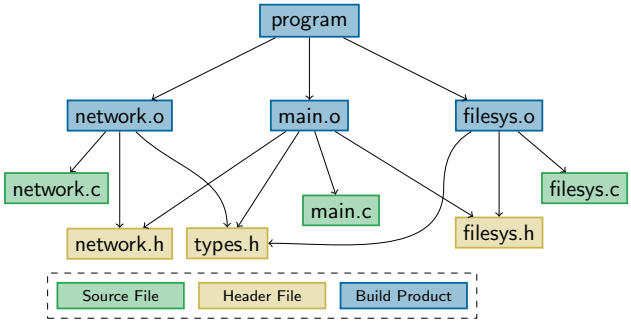
program: $(objects)
        cc -o program $(objects)

main.o: main.c types.h network.h filesys.h
        cc -o main.o -c main.c

network.o: network.c network.h types.h
        cc -o network.o -c network.c

filesys.o: filesys.c filesys.h types.h
        cc -o filesys.o -c filesys.c
```

(a) Makefile describing the build process and its dependencies.



(b) The corresponding Make-internal build-dependency graph.

Figure 1: A minimal example of a software project written in C and built with Make.

to avoid with incremental building in the first place. However, the test itself also has to be fast – it gets executed for every source module on every build. If the overhead to decide which source modules are part of the increment outweighs the cost of false positives, incremental building also becomes pointless. In practice, the trade-off between precision and overhead is tricky – precision often does not pay off [1]. On the other hand, Zhang et al. [34] showed that with the common timestamp-based tests performed by Make [6] and other build systems, up to 97 percent of calls to the compiler for C/C++ projects are unnecessary and have to be considered as redundant builds.

1.2 About This Paper

We present *c*Hash, an approach and compiler extension for the Clang C compiler to quickly detect if a source module is affected by some change and needs to be recompiled. Our approach combines speed and precision by analyzing the effect of a change on the level of the (hashed) abstract syntax tree (AST) of the program. Compared to existing state-of-the-art techniques, such as CCache, we can significantly reduce the number of false positives (by 48.18 %) at only moderate extra costs, resulting in up to 23.16 percent shorter build times for incremental builds. In particular, we claim the following contributions:

- Efficient and reliable detection of semantic source-code changes on the language level.
- Open-source implementation of the *c*Hash concept as a plugin for Clang C compiler.
- Detailed evaluation on six open-source projects and comparison with the state-of-the-art CCache tool.

The remainder of this paper is structured as follows. In Section 2 we analyze the problem of redundant builds with a special focus on C projects. In Section 3, we describe the *c*Hash approach and discuss its implementation briefly in Section 4. We evaluate *c*Hash on a set of six open-source projects in Section 5 and discuss the previous work in Section 6. Besides a discussion of our results, we also elaborate on possible threats to the validity of our findings in Section 7 and conclude the paper in Section 8.

2 Problem Analysis

All modern build systems, whether they are implemented in Make [6] or use a more sophisticated toolchain [9, 4], try to reduce the number of redundant builds in order to achieve fast incremental rebuilds. However, the employed mechanisms often fail to detect non-essential changes precisely. For example, if a developer updates the modification timestamp of the file `convolute.h`[2] from the CPython source-code repository, the build system takes 15.9 s to rebuild the entire project on our server described in Section 5.3, about half of the time that is required for a fresh build. In this scenario, all build operations were redundant, so we should not have spent time on invoking the compiler at all. With *c*Hash, we can cut down the rebuild time in this particular case to 0.72 s, a decrease of 95.5 percent.

2.1 Modular Decomposition

Incremental rebuilds are enabled by the decomposition of software into modules, which is already around since the 1970s [18]. While modules are a necessary means for the separation of concerns on the logic level, they are also often physically separated into different files. Integral to modular decomposition is the export and import of interfaces to define whether others can use a particular field or data type and if they can invoke a specific functionality. For incremental builds, modularization entails the advantage that an interface is logically split into declaration, implementation, and invocation of the interface. Hence, an invoking module is only required to be recompiled if

[2] Full path: `Modules/_decimal/libmpdec/convolute.h`

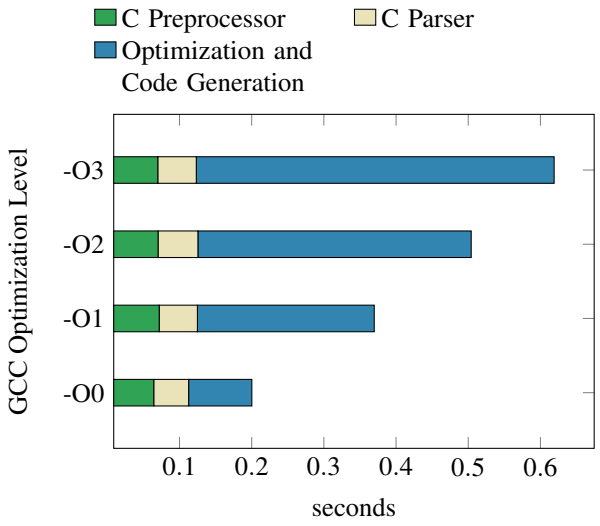


Figure 2: Average run time of the compiler phases. We compiled CPython with GCC on different optimization levels (-O0: no optimization, -O3: heavy optimization) and recorded the run time of the different compiler phases (n=850 compiler invocations per variant).

the declaration of an imported and used interface has been changed, which avoids many sources of redundant and costly recompilation [33, 30, 28].

While many languages, like Haskell or Rust, have built-in module support, the widely used C programming language lacks this feature. In C projects, modules are implemented purely idiomatically by means of the C preprocessor (CPP) and the file system. Importing another module's interface is realized via the `#include` directive of the CPP, which textually replaces `#include` directives with the content of the included file. Exporting an interface is realized via the file system by explicitly exposing the declarations (i.e., interfaces) in the corresponding header. Consequently, module dependencies in C can only be defined on granularity of files.

2.2 Build Systems and Dependencies

Since C and C-like programming languages are widely spread, their file-system–level implementation of modules heavily influenced build systems. For example, Figure 1 depicts the structure of a typical software project written in C and its build system implemented in Make [6]. Logically, the program is decomposed into the three modules `main`, `network` and `filesys`. On the file-system level, the modules are further scattered across different source files: For the `network` module, the interface declaration is located in `network.h`, while the actual implementation lives in `network.c`. Furthermore, the `network` module also imports the `types.h` definition file. From these source-code artifacts, build system and compiler generate the object file `network.o`, which is finally linked into the executable *program* file.

The developer describes all build products, their dependencies, and the production rules in the Makefile (see Figure 1a). During the build process, Make parses the Makefile and internally builds the dependency graph (see Figure 1b). The dependency graph is traversed bottom up and for each node Make checks whether the production rule has to be executed. For a clean build, all products are missing and, therefore, all rules must be executed, while for an incremental build Make examines the modification timestamp to detect out-of-date build products.

2.3 Detecting Redundant Compilation

With a *timestamp-based* method to detect redundant builds, like employed by Make, the build system compares the modification timestamp of the dependencies and the (already present) build product. If any prerequisite is newer, the production rule is re-executed to generate an updated build product, which again leads to the rebuild of all dependent build products. While this mechanism is reliable and fast, it leads to many false positives, as it is insensitive to the actual file contents. Thus, updating a file's modification time stamp suffices to cause a (cascading) recompilation.

A perfectly precise build system would only schedule an object file for recompilation, if the production rule will yield an altered binary representation. Modifying a comment, for instance, has no effect on the binary since the CPP removes comments from the token stream before compilation. However, even the introduction of a new identifier, such as a type or constant, will have no effect on the binary if the new element is not referenced by the module. Nevertheless, for such an ideal recompilation predictor, the whole compilation process would have to be done to the full extent in the same environment (e.g., optimization level). Since that would bear no benefit, various heuristics are used instead in practice, which we describe as follows.

Let's assume a given source file *C* uses a function *func* declared in header *H*. There are several possible heuristics to decide if *C* must be recompiled, each addressing a different abstraction level of the source module:

(1) The *metadata* of *H* or *C* has been changed

(2) *H* or *C* has been changed *textually*

(3) *H* or *C* has been changed *syntactically*

  (a) Syntactical change on the preprocessor level

  (b) Syntactical change on the language level

(4) The *declaration* of *func* in *H* has been changed

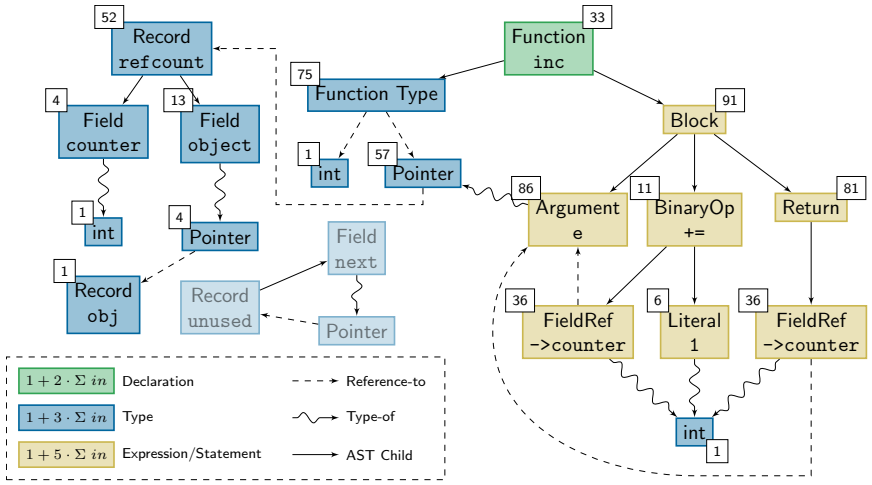
```c
struct unused {
    struct unused *next;
};

struct obj {};

struct refcount
{
    int counter;
    struct obj * ptr;
};

int
inc(struct refcount *e)
{
    e->counter += 1;
    return e->counter;
}
```

(a) C Source Code



(b) Simplified AST with annotated hash values.

Figure 3: The *c*Hash Approach. From the source module, the (standard) parser builds the abstract syntax tree and the semantic analysis establishes cross-tree references for types and variables. *c*Hash calculates a recursive hash over the AST to create a unique fingerprint of the program. In this example, we use three simplified hash rules depending only on the node class (represented by the node's color) and always take the modulo 100 from the result. Note that the `unused` record declaration is irrelevant for the resulting AST hash.

In this schema, Make and other build systems usually apply heuristic (1) by means of timestamp-based recompilation. In order to reach heuristic (2)-(4), we have to gather detailed information about the source module and its modifications. However, whether a more precise heuristic is desirable depends on the ratio of two run times: the time required to execute the redundant-build detection and the time saved for avoiding the compilation. A commonly used tool in this context is CCache [35], which uses heuristic (3a): CCache calculates a textual hash (i.e., MD4 hashes) over the preprocessed source code and uses this fingerprint as an index into an object-file cache of previous compilation results. The wide-spread adoption of CCache can be explained by the run-time proportions of the different compiler stages (see Figure 2). Compared to the whole C compiler invocation, the preprocessor takes up only 13.94 percent of the run time for the commonly used optimization level `-O2`.

In contrast to pure preprocessing, preprocessing and parsing takes only slightly longer (24.9 % of the whole invocation, `-O2`). Hence, with a more precise fingerprint of the parsed input we unlock potential higher build-time reductions for incremental compilation. In this paper, we present the *c*Hash approach, an incremental-build acceleration that applies heuristic (4) by means of hashing the abstract syntax tree within the compiler.

3 The *c*Hash Approach

Technically, *c*Hash operates similarly to the CCache tool. Both intercept the compilation process, calculate a hash value over the input, search in a cache for a previous compilation result associated with the same hash, and stop the compilation if the search was successful. Nevertheless, *c*Hash differs from CCache in two important regards: (1) *c*Hash calculates a hash over the abstract syntax tree (AST), while CCache hashes the preprocessed source code textually. (2) CCache only has to perform preprocessing, while *c*Hash must perform preprocessing, parsing, and semantic analysis. Since *c*Hash operates on the language-level instead of the CPP-syntactical level, we can easily avoid hashing of syntactic and semantic constructs (e.g., an additional declaration) that will surely not influence the compilation result. The underlying assumption is that the additional overheads of parsing and semantic analysis are only minor compared to the possible savings regarding redundant compilations, especially if a higher optimization level is chosen (see Figure 2) .

The abstract syntax tree is *the* central data structure compilers use during parsing and semantic analysis of a program. Its nodes represent the language entities (e.g., statements, expressions, types, . . .) that were identified by the parser, while the tree edges represent their syntactic nesting (e.g., statements within a function). After parsing, the semantic analysis checks the program for errors and introduces cross-tree references between semantically related nodes. For example, all variable-

definition nodes carry a reference to their respective type node. So, if we also consider these references, the AST effectively becomes a directed graph.

Figure 3a shows an example C source module with one function and three record definitions. Figure 3b depicts the corresponding (simplified) AST for the module (we omitted the root node and duplicated the `int` type). In our AST, three different classes of nodes are present: definition, statements/expression, and type nodes. For example, the function definition (`inc`) has a signature type, which itself references other types, and a compound block node that includes all statements from the function body. Furthermore, through the cross-tree references, cyclic structures can occur for recursive type definitions (see `struct unused`).

In a nutshell, *c*Hash operates directly within the compiler after the semantic analysis. In a depth-first search, we start from all top-level definitions and calculate a hash value over the semantically-enriched AST. For each node, we combine the hash values of all referenced nodes and all important node-local properties that influence the compilation into a new hash value. However, since we operate on a directed graph, nodes can be referenced more than once and two situations can arise: (1) If we have visited the node before and already calculated a hash value, we reuse it. (2) If we are currently visiting the node and encounter it again, we have detected a cycle and use a surrogate hash value instead to avoid an endless recursion. As a surrogate hash value, we use a textual representation of the type name in order to avoid collisions. This is necessary, since mutual referencing of types is possible:

```
struct x { struct y* link1; }
struct y { struct x* link2; }
```

If our surrogate value would be constant, the hash for the type `struct x` would be unchanged, if we make `link2` of type `struct *y`. In both ASTs, the depth-first search would visit the sequence (`struct x` $\rightarrow$ `link1` $\rightarrow$ `struct y` $\rightarrow$ *surrogate(link2)*). Therefore, the surrogate value must depend on the type of `link2`.

After the depth-first search, the hash is not only a fingerprint of the program semantics, but it also covers only elements that are reachable from the top-level definitions.

For illustration purposes, we executed a simplified version of the AST hashing (see Figure 3b) and annotated the intermediate hashes at the visited nodes. Here, we use a very simplistic hashing rule that incorporates only the node class (Declaration, Type, and Expression) as a node-local property. To keep the numbers small, we always calculate the modulo 100 of the result. For the top-level definition node (`inc`), the hash value calculates as follows: We add the hashes of all referenced nodes $(75+91)$, apply the node-class rule $(1+2*(166))$, and get a hash value of 33. However, not all nodes influenced this top-level hash: The record type `unused` and its children were never referenced and therefore are not covered by the top-level AST hash.

In practice, we have to be very careful when calculating the AST hash. If we omit an important property, two programs that are semantically different will end up having the same AST hash and, therefore, would be considered equal. Furthermore, we have to include all compiler flags that can influence the compilation result. In order to be on the safe side, we textually include all compiler flags into the top-level hash. If consistency between compiler upgrades is desired, we also must consider the compiler version. We also have to choose a sufficiently good hash function to avoid hash collisions. Since we are not defending against an evil attacker, we choose the efficient but non-cryptographic MurMur3 [2] hash function.

With the AST hash as a fingerprint of the source module, we can search for previous compilation results in a cache and abort the compilation process if we were successful, thereby avoiding the costly compiler phases of optimization and assembling. If the AST hash was not found, we continue the compilation process and copy the result to the cache for future invocations.

4 Implementation

We implemented the *c*Hash approach as a CLang [3] plugin for the C programming language. CLang is the C/C++/Objective-C/C++ front end of the LLVM [14] project. CLang only performs parsing and the semantic analysis of programs and hands the results, in form of LLVM intermediate representation (IR), over to LLVM for optimization and actual code generation.

The CLang plugin interface allows us to load a shared library into the compiler that is called during the compilation process. We instructed CLang to invoke *c*Hash after the semantic analysis, but right before the IR is generated and handed over to LLVM. At this point all cross-tree AST references are established, while optimizations and code generation are yet to come.

After an actual compilation, we store the AST hash for the source file and the object file for future compilations. On the next compiler invocation for the same source file, we calculate the AST hash again, compare it to the stored hash, and, in case of equality, hard link the last object file to the correct location and terminate CLang by calling `exit()`. The hard link is necessary as some build systems (CMake) remove the old object file before invoking the compiler. In contrast to the CCache tool, our current implementation stores only the last compilation result for every file instead of all previous ones. For an incremental application scenario, this has a minor influence if a change is reverted to a previous revision.

Project	Version	SLOC	Source Files		Build System	Opt. Level
			`.c`	`.h`		
LUA	5.3.4	18 k	35	26	Make	`-O2`
mbedTLS	2.4.1	56 k	123	83	CMake	`-O2`
musl	7597fc25	73 k	1322	16	Make	`-Os`
bash	4.4-p5	103 k	253	117	AutoConf	`-O2`
CPython	3.7-α1	403 k	324	325	AutoConf	`-O3`
PostgreSQL	e72059f	742 k	1152	747	AutoConf	`-O2`

Table 1: Summary of the evaluated C project repositories.

5 Experimental Results

For the evaluation, we validated the correctness of our implementation and quantified the influence of *c*Hash on the run time of incremental rebuilds. We chose six open-source projects to cover a wide range of possible application scenarios and applied *c*Hash in two typical usage scenarios. We compare our results to the CCache [35] tool (version 3.2.4) and quantify our absolute and relative prediction precision.

5.1 Evaluated Applications

We used a set of six real-world open source C projects (see Table 1) for our evaluation. This set of software projects covers a broad range of possible project properties, since they vary in size, application domain, and the employed build system.

The probed source-code bases range from small projects, like the LUA [11] language interpreter, which is mainly developed by one person, to large multi-decade, multi-person projects, like the CPython language interpreter [20]. They also differ in their usage of C language extensions: While some projects, like the musl C library [38], aim for portable and simple-structured code, others, like the mbedTLS SSL library [37] use compiler-specific features (e.g., vector types) to achieve a higher performance. The examined projects also develop at different speeds: While the PostgreSQL [19] repository lists 40 000 changes for 20 years of development, the bash command-line interpreter [7] reaches only 128 for the same period.

Furthermore, the projects employ different build systems: Small projects, like LUA and mbedTLS, often stick to plain GNU makefiles [8] and encode their dependencies manually. Larger projects often use configuration systems like CMake [4] or GNU AutoConf [9] that act as makefile generators. All examined build systems compare timestamps to detect compilation results that have to be rebuilt from the source files.

The musl C library deserves special mention, since their build system ignores some actual dependencies on purpose. Their manually encoded dependencies exclude all exported header files (515 files), since changing them would break the library's binary interface, which by definition is immutable. Hence, we exclude the public header files from our evaluation and treat them as unchangeable.

5.2 Validation of *c*Hash Implementation

As a first step, we validated the robustness of our *c*Hash implementation. In our targeted scenario, a robust implementation produces equal AST hashes for two inputs iff the compilation result is also equal.

For the validation, we built 2368 changes taken from the development history of the musl library independently. For every change and every object file, we recorded the AST hash, a textual hash of the object file, and a run-time report of the compiler-internal phases.

Over all examined changes, the compiler ran 5.68 million times and emitted 13 199 different object files. Our implementation proved to be correct and no AST hash was associated with more than one object file.

A perfectly precise predictor would exactly produce one fingerprint for every object file. For *c*Hash, we collected 55 829 different AST hashes over all changes, which results in a ratio of 1 object file : 4.23 AST hashes. Through manual investigation, we traced back 55.3 percent of the AST hashes to only 16 changes. While some of them included a major source-code reorganization, 11 changes caused a new AST hash for every source file, since they modified the compiler flags (e.g., enabled a new warning). Without these compiler-configuration changes, the ratio of object files and fingerprints drops to 1 : 2.5.

From the collected data, we could also confirm that the run-time impact of *c*Hash is minimal. On a 16-core Intel i7-2600 @ 3.4 Ghz, *c*Hash needed 9 ms on average for the AST-hash calculation. In comparison, the parser took an average 187 ms, while the rest of the compilation (i.e., optimizer, assembler) executed in 1082 ms.

5.3 Rebuild with Minimal Changes

Our first end-to-end evaluation resembles a typical scenario for an individual developer. In an already built working copy of the source code (all build products are up to date), the developer makes a *minimal change* and instructs the build system to update all products. The duration of this rebuild cycle is critical for the developer, since it is often hundreds of time a day.

We used two methods to introduce an artificial minimal change to a single file: (1) As *the* most minimal possible change, we set the modification timestamp to the current time to mimic the editor's save command. (2) To mimic a minimal textual change, we introduce a `#line 1` directive to the beginning of the file, which is left alone by the CPP and has no influence on the debug information. While both modification are surely artificial, they will

			Updated Timestamp			Textual Change		
Project	Initial Build	#files	Baseline	CCache	*c*Hash	Baseline	CCache	*c*Hash
LUA	2.03 s	61	1.09 s	−67.3 %	−59.5 %	1.10 s	16.4 %	−59.6 %
mbedTLS	3.57 s	204	1.33 s	−24.1 %	−4.1 %	1.33 s	18.9 %	−4.3 %
musl	14.29 s	1338	0.86 s	−20.6 %	−4.5 %	0.86 s	17.6 %	−4.7 %
bash	6.06 s	370	1.49 s	−70.9 %	−65.8 %	1.48 s	−9.2 %	−65.3 %
CPython	34.30 s	649	8.16 s	−77.7 %	−63.7 %	8.22 s	−24.7 %	−64.1 %
PostgreSQL	61.35 s	1891	3.16 s	−65.3 %	−42.2 %	3.12 s	8.6 %	−41.8 %

Table 2: Average rebuild duration after a minimal change. In a built working copy of the examined project, we repeated a modify–rebuild cycle for every file and measured the duration of the recompilation. Baseline shows the arithmetic average over the rebuild times, relative percentages are in respect to the baseline (n=#files).

result in an unchanged object file. Therefore, this is a best-case scenario for recompilation avoidance, since all compiler invocations are actually redundant.

We repeated the modify–rebuild cycle for every source file (headers included), measured the required rebuild time on a 48-core AMD Opteron 6180 system with 64GB of memory running Ubuntu 16.04.1, and calculated the arithmetic-average rebuild time. While our test system is an older server system, its performance is comparable to a modern developer work station. For all experiments, we instructed the build system to utilize all cores (`make -j48`). Since the initial build was done just before the actual experiment, all files were served from the main memory and disk contention was no issue.

The results are listed in Table 2. As an orientation, we also measured the time to build the project from a fresh checkout after the build system is set up. The second column holds the number of files for which the modify–rebuild cycle was executed.

For updated timestamps, CCache outperforms *c*Hash in all cases, since it only has to invoke the preprocessor and skips all subsequent compilation steps. In contrast, *c*Hash must at least wait for the parser and the semantic analysis to start calculating the AST hash.

If we introduce a textual change, CCache cannot detect the redundant rebuild and the build-time improvement diminishes or even turns negative, since CCache still has to pay the cost of maintaining the object cache. For the modification scenario, the improvements for *c*Hash remain stable and we achieve a maximum rebuild-time reduction of −65.3 percent for bash.

However, for two projects (mbedTLS and musl), we see a much smaller influence of CCache and almost no improvement by *c*Hash. Both projects have a very sparse dependency structure where a change to a source file often leads only to a single compiler invocation, while the majority of the time is spent in the linking process.

5.4 Rebuild with Commit-Sized Changes

Our second evaluation scenario resembles a usage pattern that is found in continuous-integration systems. Source-code changes are uploaded to a build server and automatically integrated into the mainline repository. For every incremental change, the build server verifies that the source code compiles and informs the developers about compilation errors. This scenario is distinct from the former one, since an uploaded change reflects the condensed editing effort of a single developer over a time period.

For this evaluation, we selected the last 500 (127 for bash) non-merge changes from the source-code repositories of the examined projects. We excluded all changes that were broken in the original repository and failed to compile. For every change, we set up the working copy to the previous (parent) change and built the project as a starting point. After applying the change, we measured the recompilation time on the same 48-core Opteron that was used for the previous scenario and calculate the arithmetic average over all non-failing changes. We repeated the evaluation for the unmodified baseline build system, CCache, *c*Hash, and a combined variant (CCache+*c*Hash). We recorded the build times, as well as the number of detected redundant builds, which we will call "hits" for brevity. The summarized results can be found in Table 3.

Our largest improvement for a single change occurred in the change `90d3da11c9` in PostgreSQL that fixes a spelling mistake in a comment located in a central header. Normally, the rebuilding of this change takes 15.6 s. While CCache correctly identifies the situation, its cache-maintenance overhead keeps the recompilation time at 3.5 s. With the compiler-internal approach of *c*Hash, we only require 2 s (−87.4 %) to rebuild.

Over all projects, and all examined changes, *c*Hash aborted the compilation in 79.75 percent of all invocations and decreased the average build time by −29.64 percent. For CPython, we even achieved an improvement of more than 50 percent. In contrast to that, CCache has a much

	Changes		Baseline		CCache		*c*Hash		CCache + *c*Hash	
	OK	Fail	Time	#Invoc.	Time	#hits	Time	#hits	Time	#hits
LUA	479	21	2.14 s	16765	−38.8 %	13761	−49.3 %	15748	−46.7 %	15748
mbedTLS	498	2	2.13 s	36654	−20.7 %	24124	−7.3 %	25750	−21.6 %	26764
musl	500	0	1.25 s	28655	−3.8 %	19587	0.7 %	19457	−3.2 %	23104
bash	108	19	2.88 s	1931	−11 %	326	−22.7 %	1281	−16 %	1354
CPython	500	0	8.27 s	20338	−46.4 %	14551	−51.4 %	19102	−53.7 %	17859
PostgreSQL	498	2	5.63 s	25934	−11 %	7184	−31.6 %	22209	−25.3 %	20909

Table 3: Rebuild time for the last 500 non-merge changes. For every change, we prepared a fully built working copy with the previous (parent) change. After applying the change, we measure the rebuild duration, as well as the number of detected redundant build operations. For the baseline, we give the arithmetic average over the time required to build one change and the number of compiler invocations. For the modified build processes, we give the change in average build time (n=#OK changes) and the accumulated numbers of detected redundant builds (#hits, higher is better).

lower hit ratio (61.05 %) and could decrease the average rebuild time by only −23.63 percent. From the total number of hits, we can quantify that *c*Hash's semantic fingerprint is at least 30.19 percent more precise than CCache's textual one.

Similar to the results of the previous scenario (Section 5.3), we see little influence of *c*Hash on the rebuild times of musl and mbedTLS. For musl, we even have a small decrease in performance due to its dependency structure. Since the developers intentionally omitted almost all dependencies on header files, the majority of compiler invocations is caused by a modified source file, which almost always involves a semantic modification.

We also combined CCache and *c*Hash by chaining their execution: First, CCache searches for the textual hash and, if unsuccessful, hands over the preprocessed code to *c*Hash. In the last two columns of Table 3, we see the result of the combined experiment. However, in our implementation, CCache interferes with the operation of *c*Hash such that the hit number decreases (CPython, PostgreSQL): CCache includes a deeper history of previous builds. If a compilation is aborted by CCache due to its cache, *c*Hash is not invoked and can, therefore, not fill its shallow cache, which is only one object file deep. If the respective source file is then modified in a way that the textual hash changes but the AST hash remains the same, *c*Hash is not able to detect the redundant build because its cache is empty. Nevertheless, the combination of CCache and *c*Hash comes close or even exceeds the best result of both methods if applied in isolation. With a combined caching strategy, this interference between CCache and *c*Hash could be avoided.

6 Related Work

As building software is an important part of the development process, attempts to reduce the build time are numerous and focus on different aspects and phases of the process. Since multiple C/C++ compilation units can be built independently, the process can be distributed over several machines. The free `distcc` [36] tool acts as a compiler wrapper and sends the preprocessed source code over the network for remote compilation. Microsoft's in-house build service CloudBuild [5] employs the same technique and distributes 20000 builds per day on up to 10000 machines and attaches to various build systems. CONCORD [21], which is Microsoft's internal alternative to CloudBuild, also uses distributed builds and speeds up the Windows build process by up to 100 times. Google's build infrastructure [12] relies on reproducible builds and distributes the work over thousands of machines.

A too coarse-grained module structure often leads to redundant builds if one of the central "God" modules are touched. Therefore, Yu, Dayani-Fard, and Mylopoulos [33] proposed a technique to refactor large header files into multiple smaller ones and thereby achieved a speedup of the compilation by nearly 32 percent. However, with higher optimization levels their speedup dropped to 12 percent. Furthermore, automatic restructuring of headers can be in conflict with the developers' intentions.

Morgenthaler et al. [17] proposed the CLIPPER tool, which automatically finds build system targets that are too coarse-grained and aids the developer in removing them. Vakilian et al. [30] examined build-system dependencies and found that nearly 50 percent of the 40000 build targets of a Google internal Java library are too coarse-grained and can be further refined. However, Miller [16] discusses that incomplete dependency graphs, which can stem from the usage of recursive GNU make systems, can yield incorrect compilation results and render incremental builds useless. Developers then often fall back to compile the software always from scratch.

Besides general build-system organization, several researchers proposed techniques to speed up the compiler invocation itself; *c*Hash being one of them. Often these propositions focus on a single compiler phase and are composable. Pre-compiled headers are an old technique

that was already used for Mesa [10, 26] and NeXT [15] and is still employed in industry [13]. For this technique, header files are translated to an intermediate format which then can be loaded faster by the compiler for all subsequent invocations.

The CCache [35] tool intercepts the compilation after the preprocessor and calculates a textual hash over the preprocessed code to detect redundant builds. For distributed build services the problem of redundant builds becomes especially severe, since many developers start compile jobs for very similar code bases. Therefore, both Microsoft and Google use textual hashing in their build services [32, 5]. However, the employed hashing method is orthogonal to the caching method and *c*Hash could act as a drop-in replacement for the textual hashes in these systems. Actually, Google's build system allows the integration of language-specific hashing methods.

For languages with a module system, Tichy [29] proposed the smart recompilation approach, which was extended by Schwanke and Kaiser [22] and Shao and Appel [24]. For smart recompilation, each exported module interface is annotated with a version stamp. Dependent modules only have to be recompiled if one of their imported interfaces has an updated version stamp. By incorporating only referenced declarations, *c*Hash achieves the same effect for languages without a proper module system. Furthermore, *c*Hash does not only include the called function signatures, but also ignores all syntactic changes that are removed by preprocessor and parser.

Zhang et al. [34] introduce the ABC tool to generate an additional unoptimized object file for each compilation unit. For each compilation unit, ABC invokes the compiler without optimizations and aborts the subsequent, but more expensive compilation process with optimizations if the unoptimized object remains unchanged. However, in case of a redundant build, *c*Hash pays only the price for parsing, while ABC has to finish the compilation.

Whaley [31] uses dynamic execution profiles to determine rarely executed code regions in Java programs, which can either be excluded from optimization or entirely from compilation. During the execution, the program falls back to using unoptimized code or even an interpreter solution. Suganuma, Yasue, and Nakatani [25] used a similar approach for dynamic compilation of Java software by focusing optimization efforts only on non-rare code paths.

7 Discussion

As we have shown in the evaluation, *c*Hash provides a significant speed up for realistic usage scenarios that occur during the development of software. In this section we want to discuss threats to the validity of our results, benefits, and give hints for future work.

7.1 Threats to Validity

One threat to the validity of our experimental results is the selection of software projects we used to evaluate the effects of *c*Hash. If their code organization and/or their change-recording policies were highly favorable for *c*Hash, our results would be overly optimistic. For example, if a project had one central header file that is updated in every single change to the repository, the time savings *c*Hash induces would be optimal. However, as discussed in Section 5.1, the chosen projects cover a broad range of properties and we have not encountered such a pattern. Furthermore, our evaluation scenarios would yield totally different results for a project with this pattern.

Another threat to our experimental validity is our implementation of *c*Hash. Although we have rigorously validated our implementation (see Section 5.2) it is not verified formally. However, the examination of 500 changes from several open-source projects and over 2000 changes for musl makes us confident in the reliability of our implementation.

Currently, we implemented the *c*Hash only for the C programming language. However, the general concept is suitable for every language that can be expressed as an AST with cross-tree references, even if it includes cyclic references. If a programming language cannot be expressed as such a structure, *c*Hash cannot be applied. One prominent example is the TEX programming language: In TEX, the program flow can influence and feedback data back to the lexer and, therefore, the actual structure becomes only visible during execution. However, such languages are rare and will only be executed in an interpreter.

Furthermore, *c*Hash is only usefully applicable if the compilation process is dominated by the middle- and back-end (optimizer, code generation) and the front end is considerably fast. However, since most modern languages offer a more expressive semantic than C (e.g., Haskell, Rust), the efficient code generation and the optimizations take longer. Therefore, we are confident that *c*Hash will perform even better for these languages than for C.

Another general impediment for a wide-ranged adoption of *c*Hash is its requirement to access internal compiler data structures. If a compiler does not provide an appropriate plugin structure or is developed as a closed-source project, *c*Hash cannot be applied. Furthermore, the AST hashing must be implemented for every programming language and for every compiler, a general implementation does not exist. However, C and C-style languages are still the most prominent compiled languages [27] and both widely used free-software compiler suites (clang, GCC) include a powerful plugin interface.

7.2 Advantages of *c*Hash

Besides the apparent benefits of faster rebuilds, *c*Hash is also build-system agnostic. Similar to the widely-adopted CCache tool, *c*Hash does not require modifications on the build-system level. We, therefore, think our unintrusive approach fosters a wide-spread adoption of *c*Hash. We demonstrated this property in the evaluation, where three different build systems were handled without any modification (see Section 5).

During the compilation process, the compiler always builds an abstract syntax tree of the program and holds it in memory. This availability allows *c*Hash to be a lightweight and self-contained mechanism that is easy to test in isolation. Furthermore, the calculation of an AST hash is computationally cheap, since only one depth-first search graph-traversal is required to calculate it. During the compilation process, such traversals are already executed dozens of times.

Since the AST is a semantic representation of the program, *c*Hash is able to detect various changes that do not lead to a changed compilation result. First of all, many syntactic modifications, like comments, whitespace-changes, or even the presence of braces are not present on the AST level.

Besides the syntactical modifications, *c*Hash is also able to ignore language-semantic changes to the compilation unit. Per default, we already ignore unused declarations and type definitions, which leads to a fine-grained dependency tracking on the symbol and the type level. As mentioned in Section 6, this property of *c*Hash brings the benefits of smart recompilation [29] to the C programming language, which in other respects lacks any module support. As C projects handle modules on the file-system level, the whole interface definition of another module is included if the header file is referenced (`#include`). With *c*Hash, we narrow this import down to the actually used interfaces and consequentially detect dependencies between modules more precisely.

7.3 AST Hash Precision

The predictive power of the AST hash in regard to detecting redundant builds is determined by two factors: (1) Which AST nodes are considered during the depth-first search. (2) What attributes of the visited nodes are included into the hash. Currently, *c*Hash is conservative in both dimensions in order to avoid false-negative compiler abortions.

On the node-selection part, *c*Hash currently ignores all AST nodes which are not referenced, directly or indirectly, from the top-level definitions of a compilation unit. However, with a more complex and compiler-aware strategy, *c*Hash could also ignore other AST nodes that will not lead to changed object file (e.g., defined but unused functions marked as `static`). A normalization step – like sorting the order of local-variable definitions – is also possible.

In regard to the AST-node fields, we ignore only fields that are known to not influence the resulting code, like origin line numbers in the source code (if correct debugging information is desired, this information should be included). In order to increase the predictive power, we could furthermore exclude variable and type-name fields. With a more relaxed equivalence relation for object files, we could additionally exclude modifiers (e.g., `inline`) if they are known to have no effect on the resulting program behavior. However, every introduction of in-depth compiler knowledge increases the complexity of the hashing mechanism, which, most probably, would make *c*Hash more fragile, especially in terms of changes in future compiler versions.

7.4 Future Work

We see several directions of future work: In order to allow co-evolution of the implementation and to do more validation, we will work on integrating *c*Hash into the CLang mainline repository. This effort also includes the integration of the *c*Hash approach into other open-source compilers (i.e., gcc).

Furthermore, we plan the implementation of the *c*Hash approach for more complex languages. As a first target, we will extend our CLang plugin to the C++ programming language. There, the usage of templates and their location in header files promises huge savings by *c*Hash.

Another direction of research is the possibility of *c*Hash to provide more fine-grained information about changed definitions and language constructs. With *c*Hash, a compiler can not only detect that the whole compilation process is redundant, but also that the compilation of a single function can be skipped. For such a partial-compilation scheme, we would start the depth-first search at the function level instead of the AST's root node.

8 Conclusion

The detection of redundant builds, which can increase the throughput of the development-testing cycle significantly, is a trade-off between precision and cost. In this paper we show how this trade-off can be optimized towards higher precision at low costs by applying language-level analyses directly in the compiler. The results for our *c*Hash approach show that on average 80 percent of all compiler invocations can already be canceled after the semantic analysis. For single projects, we speed-up the recompilation process by up to 51 percent, while single changes

even compiled up to 87 percent faster. In comparison to the state-of-the-art CCache tool, *c*Hash's AST hash fingerprinting is over 30 percent more precise.

Acknowledgments

The authors thank the anonymous reviewers and our shepherd Theodore Ts'o for their feedback. This work has been supported by the German Research Foundation (DFG) under the grants no. LO 1719/3-1 and SFB/Transregio 89 "Invasive Computing" (Project C1).

The source code of cHash and the raw data for this paper are available at:
`https://gitlab.cs.fau.de/chash`

References

[1] Rolf Adams, Walter Tichy, and Annette Weinert. "The Cost of Selective Recompilation and Environment Processing". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* (1994). DOI: 10.1145/174634.174637.

[2] Austin Appleby. *SMHasher: a testsuite for distribution, collission, and performance properties of non-cryptographic hash function*. https://github.com/aappleby/smhasher, accessed 6 Feb. 2017.

[3] *CLang: a C language family frontend for LLVM*. http://clang.llvm.org, accessed 7. Feb 2017.

[4] *CMake*. http://cmake.org, accessed 7. Feb 2017.

[5] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. "CloudBuild: Microsoft's Distributed and Caching Build Service". In: *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE)*. 2016. ISBN: 978-1-4503-4205-6. DOI: 10.1145/2889160.2889222.

[6] Stuart I. Feldman. "Make — A program for maintaining computer programs". In: *Software: Practice and experience* 9.4 (1979), pp. 255–265.

[7] *GNU Bourne Again SHell*. http://www.gnu.org/software/bash/, Git repository at http://savannah.gnu.org/projects/bash/, accessed 7. Feb 2017.

[8] *GNU Make*. http://www.gnu.org/software/make, accessed 7. Feb 2017.

[9] *GNU autoconf*. https://www.gnu.org/software/autoconf/autoconf.html, accessed 7. Feb 2017.

[10] Charles M. Geschke, James H. Morris Jr., and Edwin H. Satterthwaite. "Early Experience with Mesa". In: *SIGSOFT Softw. Eng. Notes* 2.2 (Mar. 1977), pp. 138–. ISSN: 0163-5948. DOI: 10.1145/390019.808320.

[11] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. "Lua—An Extensible Extension Language". In: *Software: Practice and Experience* 26.6 (1996), pp. 635–652. ISSN: 1097-024X.

[12] Christian Kemper. *Build in the Cloud: How the Build System works*. http://google-engtools.blogspot.de/2011/08/build-in-cloud-how-build-system-works.html, accessed 7. Feb 2017. [Online; posted 18-08-2011].

[13] Tara Krishnaswamy. "Automatic Precompiled Headers: Speeding Up C++ Application Build Times". In: *Proceedings of the 1st Conference on Industrial Experiences with Systems Software (WIESS)*. USENIX Association, 2000.

[14] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. (Palo Alto, CA, USA). 2004.

[15] Andy Litman. "An Implementation of Precompiled Headers". In: *Software – Practice and Experience* 23 (1993).

[16] Peter Miller. "Recursive Make Considered Harmful". In: *AUUGN Journal of AUUG Inc* (1998).

[17] J. David Morgenthaler, Misha Gridnev, Raluca Sauciuc, and Sanjay Bhansali. "Searching for Build Debt: Experiences Managing Technical Debt at Google". In: *Proceedings of the Third International Workshop on Managing Technical Debt*. 2012.

[18] D. L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules". In: *Commun. ACM* (1972).

[19] *PostgreSQL*. http://postgresql.org, Git repository at http://github.com/postgres/postgres, accessed 7. Feb 2017.

[20] *Python*. http://python.org, Git mirror at http://github.com/python/cpython, accessed 7. Feb 2017.

[21] Wolfram Schulte. "Changing Microsoft's Build: Revolution or Evolution". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. 2016. ISBN: 978-1-4503-3845-5.

[22] Robert W. Schwanke and Gail E. Kaiser. “Smarter Recompilation”. In: *ACM Trans. Program. Lang. Syst.* 10.4 (Oct. 1988), pp. 627–632. ISSN: 0164-0925. DOI: 10.1145/48022.214505.

[23] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. “Programmers’ build errors: a case study (at google)”. In: *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 724–734.

[24] Zhong Shao and Andrew W. Appel. “Smartest Recompilation”. In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL. 1993. DOI: 10.1145/158511.158702.

[25] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. “A Region-based Compilation Technique for a Java Just-in-time Compiler”. In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*. 2003. ISBN: 1-58113-662-5.

[26] Richard E Sweet. “The Mesa Programming Environment”. In: *ACM SIGPLAN Notices*. 1985.

[27] *TIOBE Index for January 2017*. http://www.tiobe.com/tiobe-index/, accessed 06 Feb. 2017.

[28] Alexandru Telea and Lucian Voinea. “A Tool for Optimizing the Build Performance of Large Software Code Bases”. In: *12th European Conference on Software Maintenance and Reengineering (CSMR)*. 2008.

[29] Walter F. Tichy. “Smart Recompilation”. In: *ACM Trans. Program. Lang. Syst.* 8.3 (June 1986), pp. 273–291. ISSN: 0164-0925. DOI: 10.1145/5956.5959. URL: http://doi.acm.org/10.1145/5956.5959.

[30] Mohsen Vakilian, Raluca Sauciuc, J. David Morgenthaler, and Vahab Mirrokni. “Automated Decomposition of Build Targets”. In: *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. 2015. ISBN: 978-1-4799-1934-5.

[31] John Whaley. “Partial Method Compilation Using Dynamic Profile Information”. In: *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2001. ISBN: 1-58113-335-9.

[32] Nathan York. *Build in the Cloud: Distributing Build Steps*. http://google-engtools.blogspot.de/2011/09/build-in-cloud-distributing-build-steps.html, accessed 7. Feb 2017. [Online; posted 23-09-2011].

[33] Yijun Yu, Homy Dayani-Fard, and John Mylopoulos. “Removing False Code Dependencies to Speedup Software Build Processes”. In: *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*. 2003.

[34] Ying Zhang, Yanyan Jiang, Chang Xu, Xiaoxing Ma, and Ping Yu. “ABC: Accelerated Building of C/C++ Projects”. In: *Asia-Pacific Software Engineering Conference (APSEC)*. 2015.

[35] *ccache – a fast C/C++ compiler cache*. http://ccache.samba.org/, accessed 7. Feb 2017.

[36] *distcc – a free distributed C/C++ compiler system*. http://github.com/distcc/distcc, accessed 7. Feb 2017.

[37] *mbedTLS*. http://tls.mbed.org, accessed 7. Feb 2017.

[38] *musl libc*. http://www.musl-libc.org, accessed 7. Feb 2017.

Giza: Erasure Coding Objects across Global Data Centers

Yu Lin Chen[⋆†], Shuai Mu[⋆], Jinyang Li[⋆], Cheng Huang[†], Jin Li[†], Aaron Ogus[†], Douglas Phillips[†]
[⋆]*New York University,* [†]*Microsoft Corporation*

Abstract

Microsoft Azure Storage is a global cloud storage system with a footprint in 38 geographic regions. To protect customer data against catastrophic data center failures, it optionally replicates data to secondary DCs hundreds of miles away. Using Microsoft OneDrive as an example, this paper illustrates the characteristics of typical cloud storage workloads and the opportunity to lower storage cost for geo-redundancy with erasure coding.

The paper presents the design, implementation and evaluation of Giza – a strongly consistent, versioned object store that applies erasure coding across global data centers. The key technical challenge Giza addresses is to achieve single cross-DC round trip latency for the common contention-free workload, while also maintaining strong consistency when there are conflicting access. Giza addresses the challenge with a novel implementation of well-known distributed consensus algorithms tailored for restricted cloud storage APIs. Giza is deployed to 11 DCs across 3 continents and experimental results demonstrate that it achieves our design goals.

1 Introduction

Microsoft Azure Storage is a global cloud storage system with a footprint in 38 geographic regions [27]. Since 2010, Azure Storage has grown from tens of petabytes to many exabytes, with tens of trillions of objects stored [15].

To protect customer data against disk, node, and rack failure within a data center (DC), Azure Storage applies Local Reconstruction Coding (LRC) [20] to ensure high availability and durability. LRC significantly reduces the storage cost over the conventional scheme of three-way replication.

To further protect customer data against catastrophic data center failures (say due to earthquake, tsunami, etc.), Azure Storage optionally replicate customer data to secondary DCs hundreds of miles away. It is essential to the customers that even in the unlikely, albeit inevitable, event of catastrophic data center failure, their data remain durable.

Geo-replication, however, doubles the storage cost. With many exabytes at present and exponential growth projected, it is highly desirable to lower the storage cost required for maintaining geo-redundancy.

1.1 Cross-DC Erasure Coding: Why Now?

Erasure coding across geographically distributed DCs is an appealing option. It has the potential to ensure durability in the face of data center failure while significantly reducing storage cost compared to geo-replication. The same economic argument that has driven cloud providers to erasure code data within individual data centers naturally extends to the cross-DC scenario.

However, when customer data is erasure coded and striped across multiple DCs, serving read requests would require data retrieval from remote DCs, resulting in cross-DC network traffic and latency. Furthermore, the recovery after catastrophic DC failure would trigger wide-area erasure coding reconstruction. While such reconstruction can be paced and prioritized based on demand, it nevertheless requires sufficient cross-DC network bandwidth to ensure timely recovery.

Therefore, cross-DC erasure coding only becomes economically attractive if 1) there are workloads that consume very large storage capacity while incurring very little cross-DC traffic; 2) there are enough cross-DC network bandwidth at very low cost.

For the former, Azure Storage indeed serves many customers with such workloads. Using Microsoft OneDrive as an example, Section 2 illustrates the characteristics of typical cloud storage workloads and why they are ideal for cross-DC erasure coding. For the latter, recent technological breakthroughs [26, 42] have dramatically increased bandwidth and reduced cost in cross-DC networking. For example, Facebook and Microsoft have teamed up to build *MAREA*, a new fiber optic cable under the Atlantic Ocean that will come online in 2017 with 160 Tbps capacity [12, 28]. The significant advancement in cross-DC networking is now making cross-DC erasure coding economically viable.

1.2 Challenges and Contributions

This paper presents Giza, a cloud object store that erasure codes and stripes customer data across globally distributed DCs. We aim to achieve two design goals. One, Giza should guarantee strong consistency while also minimizing operation latency. The other, Giza should make full use of existing cloud infrastructure to simplify its implementation and deployment.

Since reads and writes requires cross-DC communica-

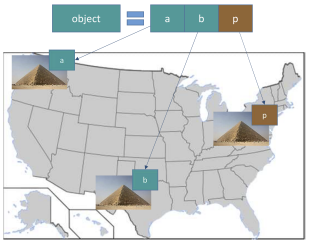
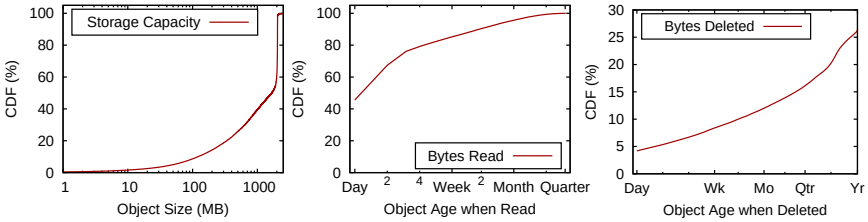


Figure 1: Storing Object in Giza



(a) **(b)** **(c)**

Figure 2: Microsoft OneDrive Characteristics

tion, latency is minimized when operations can complete within a single cross-DC roundtrip. This is possible to achieve for our target workloads (e.g. OneDrive), where objects are updated infrequently. Nevertheless, concurrent updates to the same object do exist. Furthermore, such conflicting access might originate from different DCs. Consider two concurrent requests updating the same object (with different data) from two separate DCs. Depending on network latency, the two requests may arrive at different data centers in different order. If not handled properly, this would result in data inconsistency.

To ensure strong consistency, one possible approach is to dedicate a primary DC that handles all updates and enforces execution order. However, requests from non-primary data centers have to be relayed to the primary first, incurring extra cross-DC latency, even when there are no concurrent updates. To guarantee strong consistency while minimizing latency in the common case, Giza employs FastPaxos [23], which incurs a single cross-DC roundtrip when there are no concurrent updates. When conflicting access do sometimes occur, Giza uses classic Paxos [24] and may take multiple cross-DC round trips to resolve the conflicts. We deem this to be an acceptable tradeoff.

We implement Giza on top of the existing Azure storage infrastructure. For each object, Giza stores its coded fragments in the Azure Blob storage of different DCs, and replicates its versioned meta-data containing the ids of coded fragments in the Azure Table storage of multiple DCs. Giza guarantees the consistency of versioned meta-data using Paxos/FastPaxos and it adapts both protocols to use the existing APIs of Azure Table storage.

To summarize, this paper makes the following contributions:

- We have designed and implemented Giza, a strongly consistent, versioned object store that erasure codes objects across globally distributed data centers.
- Giza achieves minimum latency in the common case when there are no concurrent conflicting access, and ensures strong consistency in the rare case under contention.
- Giza applies well-known distributed protocols–Paxos [24] and Fast Paxos [23] – in a novel way on top of restricted cloud storage APIs.
- Giza is deployed in 11 DCs across 3 continents and experimental results demonstrate that it achieves our design goals.

2 The Case for Giza

This section presents an overview of Giza, the characteristics of typical cloud storage workloads from Microsoft OneDrive, as well as the storage and networking tradeoffs exploited by Giza.

2.1 Giza Overview

Giza exploits the reduction in cross-DC bandwidth cost and leverages erasure coding to optimize the total cost of storing customer data in the cloud. It offers an externally strong consistent (linearizable [18]), versioned object store that erasure codes objects across global data centers.

Customers access Giza by creating Giza storage accounts. For each storage account, the customers have the flexibility to choose the set of data centers where their data are striped across. In addition, they can specify the erasure coding scheme. Giza employs classic $n = k + m$ Reed-Solomon coding, which generates m parity fragments from k data fragments. All n coded fragments are stored in separate DCs, which tolerates up to m arbitrary DC failures.

Figure 1 illustrates an exemplary flow of storing an object in Giza with 2 + 1 erasure coding. Giza divides the object into two data fragments (a and b) and encodes a parity fragment p. It then stores the coded fragments in 3 separate data centers.

Giza is accessible via *put*, *get*, and *delete* interface. In addition, Giza supports versioning. Each new *put* does not overwrite existing data, but rather creates a new version of the data. The old versions remain available until explicitly deleted.

2.2 Microsoft OneDrive Characteristics

Methodology: The data presented in this section is de-

rived from a three-month trace of the OneDrive service. OneDrive serves hundreds of millions of users and stores their objects which include documents, photos, music, videos, configuration files, and more. The trace includes *all* reads, writes, and updates to *all* objects between January 1 and March 31, 2016.

Large Objects Dominate: The size of the objects varies significantly, ranging from kilobytes to tens of gigabytes. While the number of small objects vastly exceeds that of large objects, the overall storage consumption is mostly due to large objects. Figure 2a presents the cumulative distribution of storage capacity consumption in terms of object size. We observe that less than 0.9% of the total storage capacity is occupied by objects smaller than 4MB. This suggests that, to optimize storage cost, it is sufficient for Giza to focus on objects of 4MB and larger∗. Objects smaller than 4MB can simply use the existing geo-replication option. This design choice reduces the overhead associated with erasure coding of small objects (including meta-data for the smaller object). As a result, all following analysis filter out objects smaller than 4MB.

Object Temperature Drops Fast: A common usage scenario of OneDrive is file sharing. Objects stored in the cloud are often shared across multiple devices, as well as among multiple users. Therefore, it is typical to observe reads soon after the objects are created. To this end, Figure 2b presents the cumulative distribution of bytes read in terms of object age when the reads occur†. It is worth pointing out that 47% of the bytes read occurred in the same day of object creation, 87% occurred within the same week, and merely less 2% occurred beyond one month. Since the temperature of the objects drops quickly, caching objects can be very effective (more below).

total reads (B) / writes (B)	2.3×	
cross-DC reads / writes with Giza	no caching	1.15×
	caching (day)	0.61×
	caching (week)	0.18×
	caching (month)	0.05×

Writes Dominate with Caching: The above table presents the effectiveness of caching. The ratio between the total amount of bytes reads to writes is 2.3×. As illustrated in Section 2.3, Giza incurs 1× and 0.5× cross-DC network traffic on writes and reads, respectively. Hence, the ratio between cross-DC traffic due to reads and writes is 1.15×. Given the temperature analysis, it is most effective for Giza to cache objects for a short period of time within one single DC. Serving reads from the caching DC dramatically reduces the cross-DC traffic due to reads. Indeed, when objects are cached for one day, the cross-DC traffic attribute to reads vs writes reduces to 0.61×. When objects are cached for one month, the ratio reduces to negligible 0.05×, in which case the cross-DC traffic is completely dominated by writes. Admittedly, caching the entire object also raises the total storage overhead to 2× (same as geo-replication) for a short period of time.

# of Versions	1	2	≥ 3
Percentage	57.96%	40.88%	1.16%

Concurrency is Rare, but Versioning is Required: The above table presents how often objects are updated and whether versioning is required. We observe that 57.96% of the objects are written once and never updated during the three-month period. For the remaining, 40.88% of the objects are updated exactly once and merely 1.16% are updated more than twice. In addition, we observe that only 0.5% of the updates are concurrent (within 1 second interval). This suggests that concurrent updates of same objects are rare in Giza (albeit possible).

Deletion is Not Uncommon: It turns out that OneDrive customers not only create new objects, but also delete old objects from time to time. To characterize how often objects are deleted and how long they have been stored upon deletion, we follow all the objects that were created during the first 3 months in 2016 and match them with object deletion trace up to one year after creation. For all the objects whose matching deletion trace records exist, we calculate the age of the objects upon deletion. Figure 2c plots the cumulative distribution of storage capacity consumption against object age‡.

We observe that a non-trivial portion of the objects were deleted within one year after their creation. These objects account for 26.5% of the total consumed storage capacity. On one hand, the amount of bytes deleted is much smaller than the total amount of bytes created, which partly explains the exponential growth of OneDrive's storage consumption. On the other hand, the percentage and amount of bytes deleted is non-trivial. This suggests that removing the deleted objects from underlining cloud storage and reclaiming capacity is crucial in achieving storage efficiency.

2.3 Giza Trade-offs

Giza offers flexible trade-offs in terms of storage cost and cross-DC network traffic, as summarized in Table 1. Although we cannot discuss the details of how Giza's trade-offs translate to overall cost reduction, our internal calculation indicates that Giza leads to savings of many millions of dollars annually for OneDrive alone.

∗Objects of tens of Gigabytes are divided into 4MB chunks before storing in cloud storage back-end.

†The analysis focuses on all the objects created during the three-month period. Hence, the object age is capped at three months.

‡The distribution curve is cut off at the right end, where the age of objects exceeds one year.

	Geo-Rep.	Giza		
# of DCs	2	3	5	7
Erasure coding	-	2 + 1	4 + 1	6 + 1
Storage overhead	2.6	1.9	1.6	1.5
Cost savings	-	**27%**	**38%**	**42%**
cross-DC traffic (put)	1x	1x	1x	1x
cross-DC traffic (get)	0	0.5x	0.75x	0.83x
DC rebuild	1x	2x	4x	6x

Table 1: Giza Trade-offs

Storage Cost: To tolerate single DC failure, geo-replication incurs the storage overhead of $2 \times 1.3 = 2.6$ (with single DC storage overhead at 1.3 [20]). With $k+1$ erasure coding, where k ranges from 2 to 6, Giza reduces the storage overhead to between 1.9 and 1.5, increasing cost savings from 27% to 42%. The storage cost savings come with inflated cross-DC traffic, examined below.

Cross-DC Traffic: For writes, Giza consumes same cross-DC traffic as geo-replication. With $k+1$ erasure coding, an object is encoded into $k+1$ fragments, where one fragment is stored in a local DC and the rest k in remote DCs. Hence, the ratio between cross-DC traffic and object size is $k/k = 1$, same as geo-replication. For reads, however, Giza consumes more cross-DC traffic. k fragments are required, where one is from the local DC and the rest $k-1$ from remote DCs. Hence, the ratio between cross-DC traffic and object size is $(k-1)/k$, which increases with k. In comparison, geo-replication serves reads entirely from the local DC and incurs no cross-DC traffic. However, as discussed in Sec **??**, the cross-DC read traffic can be cut down significantly with caching. Upon data center failure, Giza needs to rebuild lost data through erasure coding reconstruction, which requires k bytes of cross-DC traffic to reconstruct one byte of data. Geo-replication simply replicates every object and thus incurs 1× of cross-DC traffic.

Alternative Approach: Giza stripes individual objects across multiple DCs. This design leads to cross-DC traffic when serving reads. An alternative design is to first aggregate objects into large logical volumes (say 100GB) and then erasure code different volumes across multiple DCs to generate parity volumes [30]. Since every object is stored in its entirety in one of the DCs, cross-DC traffic is avoided during reads.

This design works great when objects are never deleted [30]. However, Giza must support deletion. Deleting objects from logical volumes (and canceling them from corresponding parity volumes) would result in complex bookkeeping and garbage collection, greatly increasing system complexity. In comparison, Giza keeps its design simple and relies on caching to drastically reduce the cross-DC traffic of reads to much lower than that of writes.

3 Design

This section presents the design of Giza, including the overall architecture, the data model, and the protocols for the *put*, *get*, and *delete* operations.

3.1 Overview and Challenges

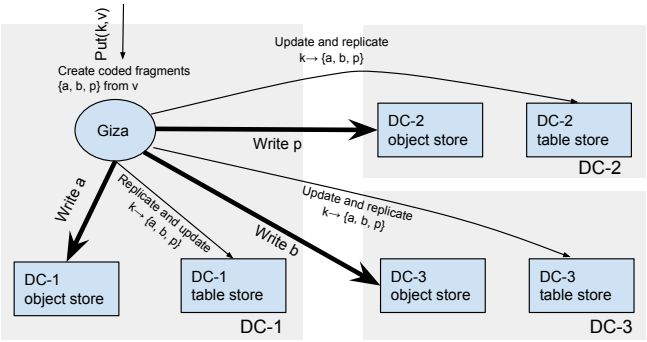


Figure 3: Giza architecture

Architecture Giza is a global-scale cloud storage system that spans across many data centers. It stores mutable, versioned objects. Figure 3 shows the architecture of Giza, which uses existing single-DC object and table stores. Giza stores an object through a *put* operation, consisting of a data operation and a metadata operation. These operations are executed in parallel to improve performance. On the data path, Giza splits and encodes the object into data and parity fragments. Each coded fragment is named by a unique identifier and stored in a different DC. Each update to the object creates a new version. The version numbers and the coded fragment IDs in each version constitutes the metadata of the object. On the metadata path, Giza replicates the metadata across the data centers.

Giza is implemented on top of the existing Azure Storage infrastructure. It stores the coded fragments in Azure Blob storage and the metadata in Azure Table storage. This layered approach provides two advantages. First, doing so allows the rapid development of Giza by reusing mature, deployed, and well-tested systems. Second, it simplifies the failure recovery and deployment: Giza runs on stateless nodes and can be readily integrated with the rest of the stateless cloud storage front-ends. Layering is commonly used in cloud infrastructure. For example, Percolator [32] supports transactions by layering over a fault-tolerant distributed table store.

Technical Challenges In Giza, each coded fragment is named by a unique identifier. As a result, fragments are immutable, which simplifies the data path.

The metadata path is more tricky, facing three main technical challenges:

1. *Building a strongly consistent, geo-replicated metadata store out of existing single-DC cloud tables.* Giza runs on stateless nodes and leverages existing well-tested cloud storage infrastructure to persist all data and metadata. The architecture simplifies development, deployment, and operation. This makes Giza quite different from other systems operating stateful servers (e.g., Cassandra, Megastore, Spanner, etc.). In addition, the cloud tables only guarantee consistency within single data center. Giza needs to orchestrate a collection of individual cloud tables across multiple data centers and achieve strong consistency globally.
2. *Jointly optimizing the data and metadata paths to achieve a single cross-DC round trip for read/write operations.* Most existing systems employ a primary-based approach, which incurs extra cross-DC round trip for secondary data centers. Giza, on the other hand, is leaderless and combines the data and metadata path in such a way that achieves single cross-DC round trips for both read and write from any data center.
3. *Performing garbage collection efficiently and promptly.* When a data object is deleted or its old versions are garbage collected, Giza must remove obsolete fragments and/or metadata from the underlying cloud blob and table storage. This turns out to be non-trivial because Giza's garbage collection mechanism must be able to handle data center failures while ensuring data consistency and durability.

3.2 Paxos using Cloud APIs

To address the above challenges, Giza adapts well-known distributed algorithms - Paxos and Fast Paxos - in a novel way on top of Azure Table.

3.2.1 Paxos and Fast Paxos in Giza: A Brief Primer

The Paxos algorithm [24] provides a mechanism to reach consensus among a set of *acceptors* and one or more *proposers*. A proposer initiates a Paxos voting process by first picking a distinguished *ballot*. All ballots are unique and can be compared to each other. The proposer sends requests and proposed values to the acceptors. Each acceptor decides whether to accept a request based on its own state. A proposed value is *committed* when it is accepted by a *quorum* of the acceptors. The acceptors update their states when a request or value is accepted.

Paxos is typically implemented via active acceptors, which are capable of comparing the ballot of incoming requests with their own states and deciding whether to accept the requests. Giza works differently and uses the cloud tables as the acceptors. It implements the acceptor logic leveraging Azure Table's *atomic conditional update* capability.

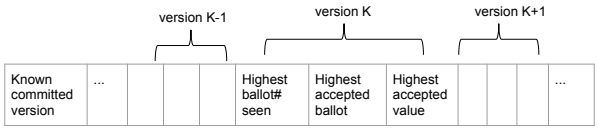


Figure 4: For each object, Giza stores the Paxos protocol state and the object metadata in a single row in the underlying cloud table.

Paxos takes 2 *phases* to reach consensus, where *phase* 1 prepares a ballot and *phase* 2 commits a value. Each phase takes 1 round trip, so applying Paxos in Giza results in 2 cross-DC round trips for the metadata path.

Fast Paxos [23] is a variation of Paxos that optimizes the performance over cross-DC acceptors. It employs two types of rounds: fast round and classic round. A fast round sends a PreAccept request and takes a single round trip to commit a value. A classic round resembles the two phases in Paxos and takes two round trips. The fast round in Fast Paxos requires a larger quorum. With 3 acceptors, a value is committed only when it is accepted by all the 3 acceptors (quorum size of 3). In comparison, Paxos is able to commit the value with 2 out of the 3 acceptors (quorum size of 2). The advantage of Fast Paxos is that when all the 3 acceptors respond, the value is committed in a single round trip. The requirement of larger quorum fits Giza perfectly, as Giza data path already requires storing fragments in 3 or more data centers.

Giza implements both Paxos and Fast Paxos. This paper discusses Fast Paxos only as its implementation requires more care (but achieves lower latency) than Paxos.

3.2.2 Metadata Storage Layout

Giza needs to persist the Paxos states together with the metadata for an object in the cloud table. We use one table row per object, with a dynamic number of columns, where each version of the object takes three columns. The layout of each table row is shown in Figure 4.

Each version is represented by a consecutive natural numbers, starting from 1. Every Giza write to the object creates a new version. For each version, Giza initiates a separate Paxos instance and uses Paxos to guard against races from concurrent writes and cloud table failures. The metadata of all versions and the states of all the Paxos instances are stored in the same table row. Specifically, the metadata contains a triplet of columns for each version (Figure 4). Two of the columns are Paxos states: `highest ballot seen` and `highest accepted ballot`. The other column, `highest accepted value`, stores the metadata, including the erasure coding scheme, the unique fragment IDs, and DCs that holds the fragments.

Giza additionally maintains a set of `known committed versions` for all those that have been

successfully committed. This is to facilitate both *put* and *get* operations, as discussed in the following sections.

3.2.3 Metadata Write - Common Case

The metadata path begins by choosing a proper new version number to initiate a Fast Paxos instance. Since version numbers need to be consecutive, the new version should succeed the most recently committed version. Giza identifies a proper version number in an optimistic fashion. Specifically, it reads `known committed versions` from the table in its local DC, then uses the next higher number as the new version number. In the uncommon case that the newly chosen version number has already been committed (but this DC missed the corresponding commit), the commit attempt would fail. Through the process, Giza learns the committed versions from the remote DCs, which allows it to choose a correct version number for retry.

Following Fast Paxos, Giza sends a PreAccept request to all the cloud tables, each located in a different DC. Each request is an *atomic conditional update* on the table row of the object. If there are no competing writes of the same object, the PreAccept request will succeed in updating the row. Otherwise, the PreAccept request will be rejected by the table and leave the row unchanged.

Whenever Giza receives a *fast quorum* of positive PreAccept responses, the corresponding version is considered to have been committed. Giza asynchronously sends a Commit confirmation to all the cloud tables to update the set of `known committed versions` to include the recently committed version. The Commit confirmation is again an atomic conditional update, which only succeeds if the version number is not yet included in the current set.

Since the Commit confirmation is completed asynchronously, the critical path only involves the PreAccept request and response. Hence, without conflict, the above described metadata write involves only one cross-DC round trip and is referred to as the *fast path*.

3.2.4 Metadata Write with Contention

The fast path may fail when Giza fails to collect a fast quorum of positive PreAccept responses. This may result from concurrent updates to the same object (contention), or because one or more cloud tables fail. In this case, Giza enters what is referred to as a *slow path* to perform classic Paxos in order to guarantee safety.

On the slow path, Giza first picks a distinguished ballot number and then replicates a Prepare request to write the ballot to all the metadata tables and wait for a majority of responses. The Prepare request is an atomic conditional update operation. The operation succeeds only if the `highest ballot seen` is no more than the ballot in the Prepare request. The operation also returns the entire row as a result.

Upon collecting a majority of successful replies, Giza needs to pick a value to commit. The rule for picking the value is categorized into three cases. In case 1, Giza looks for the highest accepted ballot in the replies. If there is one, the value from the reply is picked. In case 2, the replies contain no accepted value, but rather pre-accepted values. Giza picks the pre-accepted value returned by the maximum responses in the quorum. Both case 1 and 2 imply the possibility of an ongoing Paxos instance, so Giza picks the value so as to complete the Paxos instance first. It then starts with a new version and follows the fast path to commit its current metadata. In case 3, there is neither pre-accepted nor accepted value, which implies no real impact from contention. Giza picks its current metadata as the value and proceeds to the next steps.

Once Giza picks the value, it replicates an Accept request to all the metadata tables. The accept request is again an atomic conditional update; it succeeds in writing `highest accepted ballot` and `highest accepted value` if neither `highest ballot seen` nor `highest accepted ballot` is larger. As soon as a majority of Accept requests succeed, Giza considers the corresponding metadata write completed and sends acknowledgment to clients. Additionally, a Commit confirmation is replicated in the background, as described before.

3.2.5 Metadata Read

To get the metadata of the *latest* object version, it is *insufficient* for Giza to only read the corresponding metadata table row from its local DC. This is because the local DC might not be part of the majority quorum that has accepted the latest version. To ensure correctness, Giza needs to read the metadata rows from more than one DC.

In the common case, `known committed versions` is up-to-date and includes the latest committed version (say version k). Giza reads version k from the metadata table row in a local DC. It then confirms the lack of higher committed versions than k, from the metadata table row in a non-local DC. Hence, in the case that the metadata is replicated to 3 DCs, the metadata from 2 DCs (one local and one non-local) leads to a decisive conclusion that version k is the latest committed version. It is therefore safe for Giza to return version k to clients.

In general, Giza reads the metadata table rows from all the DCs. Whenever a majority rows have matching `known committed versions` and have not accepted any value for a higher version, Giza returns the metadata of the highest committed version.

If the replies contain an accepted value with a higher version number than the `known committed`

`versions`, Giza needs to follow a slow path similar to the one in the write operation. This is to confirm whether the higher version has indeed been committed.

3.3 Joint Optimization of Data and Metadata Operations

The naive version of Giza first writes out fragments (data and parity), and then writes out metadata, resulting in two or more cross-DC round trips. To reduce latency, we optimize Giza to execute the data and metadata paths in parallel. This is potentially problematic because either the data or metadata path could fail while the other one succeeds. Below, we describe how *put* and *get* cope with this challenge and ensure end-to-end correctness.

The *put* Operation: After generating the coded fragments, Giza launches the data and metadata paths in parallel. In the common case, Giza waits for both the data and the metadata paths to finish before acknowledging clients as well as replicating the commit confirmation. In other words, Giza ensures that `known committed versions` only include those whose data and metadata have both been successfully committed.

In one uncommon case, the data path succeeds, while the metadata path fails. Now, the fragments stored in the cloud blobs become orphans. Giza will eventually delete these fragments and reclaim storage through a cleaning process, which first executes Paxos to update the current version to *no-op*, discovers the orphan fragments as not being referenced in the metadata store, and then removes the fragments from the corresponding blob storage in all the DCs.

In another uncommon case, the data path fails, but the metadata path succeeds. This subtle case creates a challenge for the *get* operation, as addressed next.

The *get* Operation: A naive way to perform *get* is to first read the latest metadata and then retrieve the fragments. To reduce latency, Giza chooses an optimistic approach and parallelizes the metadata and the data paths.

For a *get* request, Giza first reads the metadata table row from a local DC. It obtains `known committed versions`, as well as the names and locations of the fragments of the latest version. Giza immediately starts reading the fragments from the multiple data centers. Separately, it launches a regular metadata read to validate that the version is indeed the latest. If the validation fails, Giza realizes there is a newer version. It in turn has to redo the data path by fetching a different set of fragments. This results in wasted efforts in its previous data fetch. Such potential waste, however, only happens when there is concurrent writes on the same object, which is rare.

Because the data and metadata paths are performed in parallel during *put*, it is possible (though rare) that the fragments for the latest committed version have not been written to the blob storage at the time of read. This happens if the metadata path in the *put* finishes before the data path, or the metadata path succeeds while the data path fails. In such case, Giza needs to fall back to read the previous version, as specified in `known committed versions`.

3.4 Deletion and Garbage Collection

The *delete* operation in Giza is treated as a special update of the object's metadata. When receiving a delete request (for either the entire object or specific versions), Giza executes the metadata path and writes a new version indicating the deletion. As soon as the metadata update succeeds, the deletion completes and is acknowledged.

The storage space occupied by deleted versions/objects is reclaimed through garbage collection. Giza garbage collection deletes the fragments from the blob storage and truncates the columns of the deleted versions from the metadata table row. It follows three steps: 1) fetching the metadata corresponding to the version to be garbage collected, 2) deleting the fragments in the blob storage, and 3) removing the columns of the deleted version from the metadata table row. The second step has to occur before the third one in case that the garbage collection process is interrupted and the fragments may become "orphans" without proper metadata pointing to them in the table storage.

Once all the versions of the object are deleted and garbage collected, Giza needs to remove the corresponding metadata table rows from all the DCs. This requires extra care, due to possible contention from a new *put* request. If the metadata table rows are removed brutally, the new *put* request may lead the system into an abnormal state. For instance, the *put* request could start at a data center where the metadata table row has already been removed. Giza would therefore assume that the object never existed and choose the smallest version number. Committing this version number is dangerous before the metadata table rows are removed from *all* the DCs, as this may result in inconsistency during future failure recovery.

Therefore, Giza resorts to a two-phase commit protocol to remove the metadata table rows. In the first phase, it marks the rows in all the DCs as `confined`. After this any other *get* or *put* operations are temporarily disabled for this object. In the second phase, all the rows are actually removed from the table storage. The disadvantage of this approach is obvious. It requires all the data centers to be online. Data center failure or network partition may pause the process and make the row unavailable (but can still continue after data center recovers or network partition heals).

4 Failure Recovery

Giza needs to cope with transient or permanent data center failures. Since Giza treats an entire data center as a fault domain, failures within a data center (server failures, network failures, etc...) are resolved by individual cloud object store and table store within each data center.

Transient DC failure: We broadly categorize transient DC failure to include temporary outages of the blob and table storage service in a DC. Transient DC failure may be caused by a temporary network partition or power failure. By design, Giza can still serve *get* and *put* requests, albeit at degraded performance. For example, when handling *put* requests, Giza may take more than one cross-DC round trip, because some of the DCs replicating the metadata are unavailable, resulting in fewer DCs than required for a fast path quorum.

When a data center recovers from transient failures, it needs to catch up and update the fragments in its blob storage and the metadata rows in its table storage. The process follows the Paxos learning algorithm [24]. For each object, Giza issues a read request of the metadata without fetching the fragments. If the local version matches the committed version, nothing needs to be done; if the local version is behind, the recovering process reads the fragments of all missing versions, reconstructs corresponding missing fragments and stores them in the blob storage, as well as updates the metadata row in the table storage.

Permanent DC Failure: Although extremely rare, a DC may fail catastrophically. The blob and table service within the DC may also experience long-term outages. We categorize these all as permanent DC failure.

Giza handles permanent DC failure by employing logical DC names in storage accounts. The mapping between a logical DC name to a physical DC location is stored in a separate service external to Giza. Upon a permanent DC failure, the same logical DC name is remapped from the failed DC to a healthy replacement. Giza metadata records logical DC names and therefore remains unchanged after the re-mapping. This is similar to DNS, where same domain name can be re-mapped to a different physical IP address. This way of handling failure is also reported in Chubby [8].

Upon the permanent DC failure, Giza launches recovery coordinators to reconstruct the lost fragments and re-insert the metadata rows in the replacement DC. The procedure is similar to how Giza handles transient failures yet may last longer. The reconstruction is paced and prioritized based on demand, with sufficient cross-DC network bandwidth in-place to ensure timely recovery.

	Coding	Data and Metadata DCs	Ping (max)
US-2-1	2 + 1	US(3/3)	46 ms
US-6-1	6 + 1	US(7/3)	71 ms
World-2-1	2 + 1	US(1/1), EU(1/1), JP(1/1)	240 ms
World-6-1	6 + 1	US(3/1), EU(2/1), JP(2/1)	241 ms

Figure 5: Giza Configuration (US(7/3) represents 7 DCs for data and 3 DCs for metadata, all in the US.)

5 Implementation

Giza is implemented in C++ and uses Azure Blob and Table storage to store fragments and metadata. The global footprint of Azure Storage allows for experimenting with a wide range of erasure coding parameters.

The Giza design relies on atomic conditional write. For Azure Table, we leverage its ETag mechanism. An unique ETag is generated by the table service for every write. To implement an atomic conditional write, a Giza node first reads the ETag of a table row. It then performs the condition check and issues the write request together with the ETag. Azure Table rejects the write request if the ETag in the request does not match the one in the table, which could only occur due to a concurrent write to the row.

To minimize latency, the Giza node *delegates* its conditional write requests to remote Giza nodes, which reside in the same DCs as the tables and act as proxies in reading the ETag and writing the local table row.

5.1 Experimental Setup

We run experiments using four configurations: US-2-1, World-2-1, US-6-1, and World-6-1. Figure 5 describes the data centers participating in each configuration, and the max ping latency between the DCs. Unless explicitly stated, all experiments erasure code objects of 4MB, the dominating size in our target workloads.

We also compare Giza with CockroachDB [9], an open source implementation of Google spanner. Our CockroachDB experiments use the US-2-1 configuration, as CockroachDB doesn't yet support world wide replication. In every data center, we run three CockroachDB instances for local DC replication. Each CockroachDB writes to a dedicated HDD with no memory caching. We have configured the CockroachDB instances following the recommended production setting by the CockroachDB developers. For example, we run NTP to synchronize clocks of the different CockroachDB instances.

6 Evaluation

For evaluation, we deploy Giza on top of the Microsoft Azure platform across 11 data centers (7 in North America, 2 in Europe and 2 in Asia). Giza nodes are Azure virtual machines with 16 cores, 56 GB of RAM, and Gigabit Ethernet. As describe in Section 3, all the Giza nodes are stateless. For each Giza storage account, a lo-

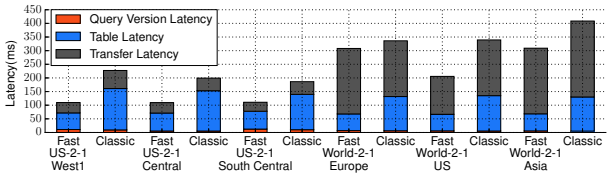
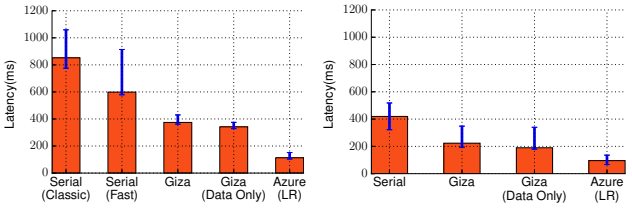


Figure 6: Fast Paxos and Classic Paxos Comparison



(a) Put **(b) Get**

Figure 7: Giza Overall Latency

cally redundant Azure Blob and Table storage account is created in every DC. Upon receiving *get* or *put* requests, the Giza nodes execute the data and the metadata paths to read or write objects.

6.1 Metadata Latency

We implement Giza's metadata path with both Classic and Fast Paxos. Here, we compare the performance of the two algorithms and examine their effects on Giza's metadata path latency. Figure 6 presents the metadata latencies and breakdowns for both US-2-1 and World-2-1 configurations. The results include running proposers in each of the DCs.

The metadata latency consists of three parts: query version latency, transfer latency, and table latency. The query version latency is determined by reading the possible highest version from the proposer's local table. This request is not part of the concensus protocol and is the same for both Fast and Classic Paxos. The transfer latency is the amount of time spent on network communication between the proposer and the furthest Giza proxy in a Paxos quorum. Here, the latency of Classic Paxos, which incurs two cross-DC round trips, is not strictly twice as much as the latency of Fast Paxos. This is because the Classic Paxos quorum is smaller than the Fast Paxos quorum. As a result, the distance between the proposer and the *furthest* proxy is smaller in a Classic Paxos quorum. The table latency is the latency for a Giza proxy to conditionally update its local DC table. Since Classic Paxos requires two rounds and hence two table updates, its table latency is twice that of Fast Paxos.

For the US-2-1 configuration, we observe that the metadata latency is dominated by table latency. In this case, Fast Paxos is much faster than Classic Paxos, regardless of the proposer's location.

For the World-2-1 configuration, transfer latency becomes a substantial part of the overall metadata latency. In this case, despite of taking two cross-DC round trips, the Classic Paxos implementation can have lower transfer latency. Nevertheless, the table latency of Classic Paxos is still twice that of Fast Paxos. As a result, the Fast Paxos implementation has lower latency, regardless of the proposer's location.

6.2 Giza Latency

The design of Giza went through multiple iterations and this section illustrates the performance gain for each iteration. For the interest of space, we focus on the World-2-1 configuration. All latency results include error bars representing the 10th and 95th percentile.

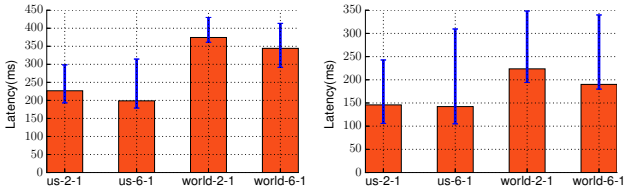
6.2.1 Giza *Put* Latency

Figure 7a shows the Giza overall *put* latency for 4MB data. We compare Giza with its two previous iterations where the metadata path is not parallelized with the data path. In the first iteration, Giza runs the data path first. After completing the data path, Giza runs the metadata path with the Classic Paxos implementation. In the second iteration, we replaced Classic Paxos with Fast Paxos, improving latency performance. Giza parallelizes metadata path with data path, which can results in extra metadata or data clean up if either path fails to complete. However, the performance gain is significant. We also included a baseline which is the time it takes a proposing data center to issue a blob store request to the farthest data center in the quorum. Finally, we include the latency for storing the 4MB data directly to Azure storage, which is locally replicated.

The results show that Giza's performance beats the other two alternatives in the common case and has closest latency to the baseline. The median latency of Giza's *put* is 374 ms, only 30 ms higher than the baseline. This is due to the latency of erasure coding 4MB data. On the other hand, the serial Paxos version takes 852 ms, and the serial Fast Paxos version takes 598 ms. In summary, the latency cost for tolerating data center failure with Giza is a little more than 3 time that of local replication.

6.2.2 Giza *Get* Latency

Figure 7b shows Giza's *get* performance comparison. The alternative design here is the non-optimistic *get* where the most current version for a blob is not assumed to be stored in the current data center. Hence, the metadata path and data path are executed sequentially, taking 419 ms. Giza's optimistic *get*, which runs the metadata path and data path in parallel, takes 223ms. Giza's *get* latency is higher than the baseline by 33 ms. The perfor-



(a) Put (b) Get

Figure 8: Performance for Giza in different setups

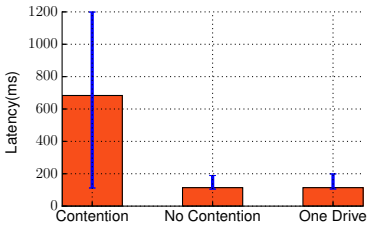


Figure 9: Contention vs No Contention

mance gap between Giza and baseline is higher because Giza needs to do a local table retrieval first before starting the datapath. In addition, it needs to decode the data fragments. Here the latency cost of erasure encoding on the read path with Giza is roughly twice that of reading from a locally redundant Azure storage.

6.3 Footprint Impact

Giza offers customers the flexibility to choose the set of data centers, as well as the erasure coding parameters (e.g., the number of data fragments k). It turns out that increasing k not only reduces storage overhead, but also overall latency. This is because the latency in Giza is often dominated by the data path. Erasure coding with a larger k results in smaller fragments and fewer bytes stored in each DC's blob storage. This reduces the data path latency and in turn the overall latency.

Figure 8a and Figure 8b present the latency impact given different Giza footprints and erasure coding parameters. All the requests are generated from US-Central. Comparing US-2-1 to US-6-1 (World-2-1 to World-6-1), it is clear that increasing k from 2 to 6 reduces the latency for both *put* and *get*.

6.4 Comparing Giza with CockroachDB

Ideally, we would like to compare Giza with an existing storage system with similar functionalities. However, there is no off-the-shelf erasure coded system. Hence, we implemented Giza on top of CockroachDB using its transaction support. To do this, we create four different tables in CockroachDB: one metadata table and three data tables (for storing coded fragments). The metadata table is replicated across all three DCs. Each of the data tables is replicated three times within its respective data center. This is to match the local replication of Azure Table within individual DCs.

We implement Giza's *put* as a transaction consisting of storing each coded fragment at the corresponding data table and storing the metadata information in the metadata table. Since CockroachDB is not optimized for storing large objects, we evaluate the performance of *put*s on 128KB objects. The median *put* latency of 128KB objects under CockroachDB is 333ms, much higher than that of Giza (<100ms).

We implement Giza's *get* as a transaction consisting of reading the metadata from the metadata table and two coded fragments from the data tables. The median *get* latency under CockroachDB is lower than that of Giza by 20%. This is because CockroachDB directly reads from local HDD, which is faster than Giza reading from Azure storage. To demonstrate this, we equalize the storage layer to substitute Azure latency with local HDD latency. Indeed, Giza's performance with equalized storage is slightly better than that of CockroachDB.

6.5 Giza Contention

Giza is optimized for low contention workloads. So, it employs a simple strategy for handling contention. In the event of contention, a Giza node that fails the fast round falls back to a classic round. In addition, Giza implements exponential back-off with the latency starting from the median cross-DC latency whenever prepare phase or accept phase further fails.

Figure 9 compares the performance of Giza driven by the OneDrive trace to that with no contention at all. In the OneDrive trace, only 0.5% of updates are concurrent (within 1 second interval). Hence, it is not surprising that the performance of Giza driven by the OneDrive trace is almost identical to that with no contention.

Figure 9 also presents the latency results of *adversary* contention. In this case, two Giza nodes within the same data center are issuing back-to-back concurrent puts to update the same object. This is definitely not the scenario that Giza targets. We include the results merely for the interest of our readers.

7 Related Work

Erasure Coding in Cluster Storage: Erasure coding has long been applied in many large-scale distributed storage systems [34, 41, 16, 1, 38, 35, 40], including productions systems at Facebook [6], Google [13, 14] and Microsoft Azure [20]. These solutions generalize the RAID approach [31, 39] to a distributed cluster setting. Giza is unique in synchronously replicating erasure coded data across WAN and minimizing cross-DC latency. In addition, Giza provides globally consistent

put and *get* with versioning support.

Erasure Coding in Wide Area Storage: HAIL [7], OceanStore [22, 33], RACS [2], DepSky [5] and NCCloud [19] all stripe and erasure code data at a global scale.

HAIL [7] is designed to withstand Byzantine adversaries. OceanStore [22, 33] assumes untrusted infrastructure and serializes updates via a primary tier of replicas. Giza operates in a trusted environment.

RACS [2] and DepSky [5] address conflicts caused by concurrent writers using Apache ZooKeeper [21], where readers-writer locks are implemented at per-key granularity for synchronization. Giza, on the other hand, implements consensus algorithms for individual keys and achieves strong consistency without centralized coordinators. In addition, Giza employs a leaderless consensus protocol. Updates may originate from arbitrary data centers and still complete with optimal latency without being relayed through a primary.

NCCloud [19] implements a class of functional regenerating codes [11] that optimize cross-WAN repair bandwidth. Giza employs standard Reed-Solomon coding and leaves such optimization to future.

Facebook f4 [30] is a production warm blob storage system. It applies erasure coding across data centers for storage efficiency. As discussed in Section 2.3, f4 avoids the deletion challenge by never truly deleting data objects. Whenever a data object is deleted, the unique key used to encrypt the object is destroyed while the encrypted data remains in the system. This simplification suits Facebook very well, because its deleted data only accounts for 6.8% of total storage and Facebook could afford not to reclaim the storage space [30]. This, unfortunately, is not an option for Giza, as our workloads show much higher deletion rate. Not reclaiming the physical storage space from deleted data objects would result in significant waste and completely void the gain from cross-DC erasure coding. Furthermore, not physically deleting customer data objects - even if encrypted - wouldn't meet the compliance requirements for many of our customers.

Separating Data and Metadata: It is common for a storage systems to separate data and metadata path, and design a separate metadata service to achieve better scalability, e.g., FARSITE [3] and Ceph [37]. Gnothi [36] replicates metadata to all replicas while data blocks only to a subset of the replicas. Cocytus [40] is a highly available in-memory KV-store that applies replication to metadata and erasure coding to data so as to achieve memory efficiency. Giza follows a similar design path, and store data in commodity cloud blob storage and metadata in commodity NoSQL table storage.

Consistency in Global Storage: Megastore [4] and Spanner [10] applies Multi-Paxos to maintain strong consistency in global databases. Both of them requires two round trips for a slave site to commit. Mencius [25] takes a round-robin approach for proposers in different sites, amortizing commit latency. EPaxos [29] uses fine-grained dependency tracking at acceptor-side to ensure low commit latency for both non-contended and contended requests. In comparison, Giza takes a refined approach based on FastPaxos [23], separating metadata and data path before committing. This design choice allows Giza to serve most requests still in single cross-DC round trip while keeping servers stateless, using the limited ability of table service. Metasync [17] implements Paxos using the append functionality provided by cloud file synchronization services such as DropBox, OneDrive. By contrast, Giza implements Paxos using conditional-write APIs of cloud tables. The latter leads to a more efficient implementation as clients do not need to download and process logs from the cloud storage in order to execute Paxos.

8 Conclusion

In this paper, we present the design and evaluation of Giza – a strongly consistent, versioned object store that encodes objects across global data centers. Giza implements the Paxos consensus algorithms on top of existing cloud APIs and have separate data and metadata paths. As a result, Giza is fast in normal operation for our target workloads. Our evaluation of Giza on a deployment over 11 DCs across 3 continents demonstrates that Giza achieves much lower latency than naively adopting a globally consistent storage system.

Acknowledgments

We thank Andy Glover, Jose Barreto, Jon Bruso, Ronakkumar Desai, Joshua Entz from the OneDrive team for their many contributions. Special thanks go to Jeff Irwin for his contributions that helped enable Giza. We also thank all of the members of the Azure Storage team for invaluable discussions and iterations, as well as Taesoo Kim and anonymous reviewers for their insightful feedback. This work was partially supported by ONR grant N00014-16-1-2154.

References

[1] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. P. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, et al. Ursa minor: versatile cluster-based storage. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*. Dec. 2005.

[2] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon RACS: a case for cloud storage diversity. In *Proceedings of ACM Symposium on Cloud Computing (SoCC)*. June 2010.

[3] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Dec. 2002.

[4] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Léon, Y. Li, A. Lloyd, and V. Yushprakh Megastore: providing scalable, highly available storage for interactive services. In *Proceedings of Biennial Conference on Innovative Data Systems Research (CIDR)*. Jan. 2011.

[5] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa DepSky: dependable and secure storage in a cloud-of-clouds. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*. Apr. 2011.

[6] D. Borthakur, R. Schmidt, R. Vadali, S. Chen, and P. Kling Hdfs raid. In *Hadoop user group meeting*. 2010.

[7] K. D. Bowers, A. Juels, and A. Oprea HAIL: a high-availability and integrity layer for cloud storage. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*. Nov. 2009.

[8] M. Burrows The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Nov. 2006.

[9] *CockroachDB*. http://www.cockroachlabs.com/.

[10] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Oct. 2012.

[11] R. G. Dimakis, P. B. Godfrey, Y. Wu, M. O. Wainwright, and K. Ramch Network coding for distributed storage systems. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*. 2007.

[12] *Facebook and Microsoft to Build Fiber Optic Cable Across Atlantic*. http://www.wsj.com/articles/facebook-and-microsoft-to-build-fiber-optic-cable-across-atlantic-1464298853. May 2016.

[13] A. Fikes Storage architecture and challenges. *Talk at the Google Faculty Summit* (2010).

[14] 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 *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Oct. 2010.

[15] A. Greenberg SDN for the cloud. In *Keynote in the 2015 ACM Conference on Special Interest Group on Data Communication*. 2015.

[16] A. Haeberlen, A. Mislove, and P. Druschel Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*. May 2005.

[17] S. Han, H. Shen, T. Kim, A. Krishnamurthy, T. Anderson, and D. Wetherall MetaSync: file synchronization across multiple untrusted storage services. In *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*. July 2015.

[18] M. P. Herlihy, and J. M. Wing Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

[19] Y. Hu, H. Chen, P. Lee, and Y. Tang NCCloud: applying network coding for the storage repair in a cloud-of-clouds. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*. Feb. 2012.

[20] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, et al. Erasure coding in Windows Azure storage. In *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*. June 2012.

[21] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*. June 2010.

[22] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Nov. 2000.

[23] L. Lamport *Fast Paxos*. Tech. rep. MSR-TR-2005-112. Microsoft Research, 2005.

[24] L. Lamport Paxos made simple. *ACM SIGACT News* 32, 4 (2001), 18–25.

[25] Y. Mao, F. P. Junqueira, and K. Marzullo Mencius: building efficient replicated state machines for WANs. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Dec. 2008.

[26] R. Mears, L. Reekie, S. Poole, and D. Payne Low-threshold tunable CW and Q-switched fibre laser operating at 1.55 μm. *Electronics Letters* 3, 22 (1986), 159–160.

[27] *Microsoft Azure Regions*. https://azure.microsoft.com/en-us/regions/.

[28] *Microsoft, Facebook to lay massive undersea cable*. http://www.usatoday.com/story/experience/2016/05/26/microsoft-facebook-undersea-cable-google-marea-amazon/84984882. May 2016.

[29] I. Moraru, D. G. Andersen, and M. Kaminsky There is more consensus in egalitarian parliaments. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*. Nov. 2013.

[30] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al. f4: Facebook's warm BLOB storage system. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Oct. 2014.

[31] D. Patterson, G. Gibson, and R. Katz A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*. June 1988.

[32] D. Peng, and F. Dabek Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Oct. 2010.

[33] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz Pond: the OceanStore prototype. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*. Mar. 2003.

[34] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence FAB: building distributed enterprise disk arrays from commodity components. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Oct. 2004.

[35] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur Xoring elephants: Novel erasure codes for big data. *The Proceedings of the VLDB Endowment (PVLDB)* 6, 5 (Mar. 2013).

[36] Y. Wang, L. Alvisi, and M. Dahlin Gnothi: separating data and metadata for efficient and available storage replication. In *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*. June 2012.

[37] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn Ceph: A scalable, high-performance distributed file system. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Nov. 2006.

[38] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou Scalable Performance of the Panasas Parallel File System. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*. Feb. 2008.

[39] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems (TOCS)* 14, 1 (Feb. 1996).

[40] H. Zhang, M. Dong, and H. Chen Efficient and available in-memory KV-store with hybrid erasure coding and replication. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*. Feb. 2016.

[41] Z. Zhang, S. Lin, Q. Lian, and C. Jin RepStore: a self-managing and self-tuning storage backend with smart bricks. In *Proceedings of International Conference on Autonomic Computing*. 2004.

[42] B. Zhu, T. Taunay, M. Fishteyn, X. Liu, S. Chandrasekhar, M. Yan, J. Fini, E. Monberg, and F. Dimarcello 112-Tb/s space-division multiplexed DWDM transmission with 14-b/s/Hz aggregate spectral efficiency over a 76.8-km seven-core fiber. *Optics Express* 19, 17 (2011), 16665–16671.

SmartCuckoo: A Fast and Cost-Efficient Hashing Index Scheme for Cloud Storage Systems

Yuanyuan Sun, Yu Hua*, Song Jiang†, Qiuyu Li, Shunde Cao, Pengfei Zuo
Wuhan National Laboratory for Optoelectronics, School of Computer
Huazhong University of Science and Technology
†*University of Texas, Arlington*
**Corresponding Author: Yu Hua (csyhua@hust.edu.cn)*

Abstract

Fast query services are important to improve overall performance of large-scale storage systems when handling a large number of files. Open-addressing cuckoo hash schemes have been widely used to support query services due to the salient features of simplicity and ease of use. Conventional schemes are unfortunately inadequate to address the potential problem of having endless loops during item insertion, which degrades the query performance. To address the problem, we propose a cost-efficient cuckoo hashing scheme, named SmartCuckoo. The idea behind SmartCuckoo is to represent the hashing relationship as a directed pseudoforest and use it to track item placements for accurately predetermining the occurrence of endless loop. SmartCuckoo can efficiently predetermine insertion failures without paying a high cost of carrying out step-by-step probing. We have implemented SmartCuckoo in a large-scale cloud storage system. Extensive evaluations using three real-world traces and the YCSB benchmark demonstrate the efficiency and efficacy of SmartCuckoo. We have released the source code of SmartCuckoo for public use.

1 Introduction

Efficient query services are critical to cloud storage systems at various scales, especially when they process a massive amount of data. According to the report of International Data Corporation (IDC) in 2014, the amount of information created and replicated will reach 44 Zettabytes in 2020 [49], and nearly 50% of cloud-based services will rely on data in storage systems [21]. Moreover, in a recent survey of 1,780 data center managers in 26 countries, over 36% of respondents face two critical challenges, which are efficiently supporting a flood of emerging applications and handling the rapidly increasing data management complexity [2]. This reflects a reality that we are generating and accessing much more data than ever and this trend continues at an accelerated pace. This data volume explosion has imposed great challenge on storage systems, particularly on their support on efficient data query services. In various computing facilities, from small hand-held devices to large-scale data centers, people are collecting and analyzing ever-greater amounts of data. Users routinely generate queries on hundreds of Gigabytes of data stored on their local disks or cloud storage systems. Commercial companies generally handle Terabytes and even Petabytes of data each day [6, 10, 54].

It is becoming increasingly challenging for cloud storage systems to quickly serve queries, which often consumes substantial resources to support query-related operations [51]. Cloud management systems usually demand the support of low-latency and high-throughput queries [7]. In order to address these challenges, query services have received many attentions, such as top-k query processing [23, 34, 37], security model for file system search in multi-user environments [9], metadata query on file systems [26, 38], Web search using multi-cores in mobile computing [27], graph query processing with abstraction refinement [52], energy saving for online search in datacenters [50], efficient querying of compressed network payloads [48], reining the latency in tail queries [22], and scaling search data structures for asynchronized concurrency [12].

An efficient hashing scheme is important for improving performance of query services. A hash table needs to map keys to values and supports constant-time access in a real-time manner. Hash functions are used to locate a key to a unique bucket. While keys may be hashed to the same bucket (the occurrence of hash collisions), lookup latency can become higher with more collisions in a bucket. Cuckoo hashing [43] is a fast and simple hash structure with the constant-time worst-case lookup ($O(\ln \frac{1}{\varepsilon})$) and consumes $(1+\varepsilon)n$ memory consumption, where ε is a small constant. Due to its desirable property of open addressing and its support of low lookup latency, cuckoo hashing has been widely used in real-world cloud applications [13, 24, 28, 35, 46]. Cuckoo hashing uses multiple (usually two in practice) hash functions for resolving hash collisions and recursively kicks items

out of their current positions. Unlike standard hashing schemes that provide only one position for placing an item, cuckoo hashing provides multiple (usually two) possible positions to reduce the probability of hash collisions. To determine the presence of an item, the cuckoo hashing will probe up to two positions, and the worst-case lookup time is a constant.

However, the cuckoo hashing suffers from substantial performance penalty due to the occurrence of endless loops. Currently, the existence of endless loop is detected only after a potentially large number of step-by-step kick-out operations. A search for insertion position in an endless loop turns out to be fruitless effort. In order to deliver high performance and improve lookup efficiency, we need to address two major challenges.

Substantial Resources Consumption. In an endless loop, an insertion failure can only be known after a large number of in-memory operations, and the penalty can substantially compromise the efficiency of cuckoo hashing schemes. When a hash table is substantially occupied, many such loops occur, which can substantially increase insertion costs.

Nondeterministic Performance. Cuckoo hashing essentially takes a random walk to find a vacant bucket for inserting an item since the knowledge on the path for this walk is not obtained in advance [19, 33]. This scheme does not leverage the dependencies among the positions of items. Before walking sufficiently long on the path, one can hardly know if an endless loop exists. Moreover, the cuckoo hashing provides multiple choices of possible positions for item insertion. The kick-out operations need to be completed in an online manner.

Existing schemes have not effectively addressed the two challenges. For example, *MemC*3 [16] uses a large kick-out threshold as its default kick-out upper bound, which possibly leads to excessive memory accesses and reduced performance. *Cuckoo hashing with a stash (CHS)* [29] addresses the problem of endless loops by using an auxiliary data structure as a stash. The items that introduce hash collisions are moved into the stash. For a lookup request, *CHS* has to check both the original hash table and the stash, which increases the lookup latency. Furthermore, *bucketized cuckoo hash table (BCHT)* [15, 44, 45, 55] allocates two to eight slots into a bucket, in which each slot can store an item, to mitigate the chance of endless loops, which however results in poor lookup performance due to multiple probes.

In order to clearly demonstrate the performance impact of endless loops, we measure the loop ratio in *CHS* with three real-world traces (experiment details can be found in Section 4.1). The loop ratio is defined as the percentage of failed insertions due to the existence of endless loops among all item insertions. Figure 1 shows that more than 25% insertions walk into endless loops at a load factor of 0.9 for the hash table, which leads to substantial time and space overheads for carrying out rehashing operations and allocating additional storage space, respectively. The load factor is the ratio of the number of occupancies to that of total buckets in the hash table. To this end, we need to mitigate and even eliminate the occurrence of endless loops to reduce the space and time overheads.

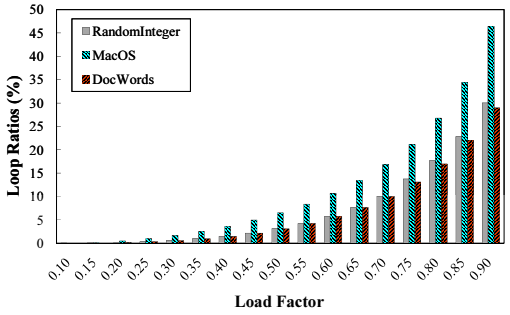


Figure 1: The loop ratios in CHS with three traces.

In this paper, we propose a cost-effective cuckoo hashing scheme, named SmartCuckoo. SmartCuckoo allows flexible configurations and fast item lookup, and achieves much improved insertion performance. Our work aims to answer the following questions: *(1) Is there a vacant bucket available for an item to be inserted before starting kick-outs on a path? (2) How to guarantee efficiency of the insertion and lookup using a space-efficient and lightweight auxiliary structure?*

SmartCuckoo leverages a fast and cost-efficient predetermination operation to help avoid unnecessary kick-out process due to endless loops. This operation runs before item insertion starts by using an auxiliary structure. Moreover, an insertion failure can be identified without any kick-out operations and manual setting of iteration thresholds. SmartCuckoo can avoid the endless loops of cuckoo hashing and deliver high performance. This paper has made the following contributions.

Cost-effective Hashing Scheme. SmartCuckoo retains cuckoo hashing's advantage of space efficiency and constant-time queries via open addressing. In the meantime, SmartCuckoo is able to predetermine insertion failures without the need of carrying out continuous kick-out operations, thus significantly reducing the insertion latency and supporting fast lookup services.

Deterministic Performance. Conventional cuckoo hashing schemes take many kick-out operations in their insertion operations before detecting endless loops and consuming substantial system resources. By categorizing insertions into different cases, SmartCuckoo helps predetermine the result of a new insertion to avoid the endless loop by leveraging the concept of maximal

pseudoforest. SmartCuckoo hence makes insertion performance more predictable.

System Implementations and Public use. We have implemented all the components and algorithms of SmartCuckoo and released the source code for public use[1]. In order to evaluate the performance of SmartCuckoo, we compared it with state-of-the-art schemes, including *CHS* [29] as the evaluation baseline, *libcuckoo* [36], as well as *BCHT* [15].

2 Background

This section presents the research background of the cuckoo hashing and the pseudoforest theory. As a cost-efficient hashing scheme, the cuckoo hashing utilizes open addressing to improve lookup efficiency for large datasets. In the cuckoo hashing, the relationship between items and buckets can be described by a cuckoo graph, where each edge represents a hashed item and its two vertices represent the positions of the hashed item in the hash directory.

Cuckoo hashing does not require dynamic memory allocation, which can be efficiently exploited to provide real-time query services. The cuckoo hashing is able to support fast queries with worst-case constant-scale lookup time due to its addressing open to multiple positions for one item.

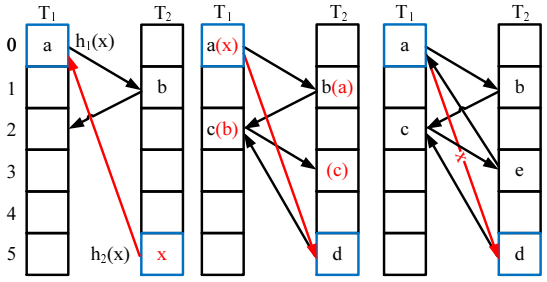
2.1 The Cuckoo Hashing

Cuckoo hashing [42, 43] is a dynamization of a static dictionary. The hashing scheme resolves hash collisions in a multi-hash manner.

Definition 1 ***Conventional Cuckoo Hashing.*** *Let d be the number of hash tables, and S be the set of keys. For the case of $d = 2$, conventional cuckoo hashing uses two hash tables, T_1 and T_2 with a size of n, and two hash functions h_1, h_2: $S \to \{0, \ldots, n-1\}$. A Key $k \in S$ can be inserted in either Slot $h_1(k)$ of T_1 or Slot $h_2(k)$ of T_2, but not in both. The two hash functions h_i (i = 1 or 2) are independent and uniformly distributed.*

As shown in Figure 2, we use an example to illustrate the insertion process in the conventional cuckoo hashing. In the cuckoo graph, the start point of an edge represents the actual storage position of an item and the end point is the backup position. For example, the bucket $T_2[1]$ storing Item b is the backup position of Item a. We intend to insert the item x, which has two candidate positions $T_1[0]$ and $T_2[5]$ (blue buckets). There exist three cases about inserting Item x:

- Two items (a and b) are initially located in the hash tables as shown in Figure 2(a). When inserting Item x, one of x's two candidate positions (i.e., $T_2[5]$) is empty. Item x is then placed in $T_2[5]$ and an edge is added pointing to the backup position ($T_1[0]$).
- Items c and d are inserted into hash tables before Item x, as shown in Figure 2(b). Two candidate positions of Item x are occupied by Items a and d respectively. We have to kick out one of occupied items (e.g., a) to accommodate Item x. The kicked-out item (a) is then inserted into its backup position ($T_2[1]$). This procedure is performed iteratively until a vacant bucket ($T_2[3]$) is found in the hash tables. The kick-out path is $x \to a \to b \to c$.
- Item e is inserted into the hash tables before Item x, as shown in Figure 2(c). There is no vacant bucket available to store Item x even after substantial kick-out operations, which results in an endless loop. The cuckoo hashing has to carry out a rehashing operation [43].



(a) Vacant bucket(s). (b) Finite kicks. (c) An endless loop.

Figure 2: The conventional cuckoo hashing data structure.

A lookup operation probes two candidate positions of an item. Buckets $T_1[0]$ and $T_2[5]$ will be probed for searching Item x, as shown in Figure 2. If the queried item is stored in the hash tables, it must be in one of its two candidate positions.

When all candidate buckets of a newly inserted item have been occupied, the cuckoo hashing needs to iteratively carry out kick-out operations to identify a vacant bucket, which possibly causes an endless loop and an insertion failure, until a kick-out path is tried and a threshold of steps on the path is reached without locating a vacant position.

2.2 Pseudoforest Theory

A pseudoforest is an undirected graph in the graph theory and each of maximally connected components, named **subgraphs**, has at most one cycle [5,20]. In other words,

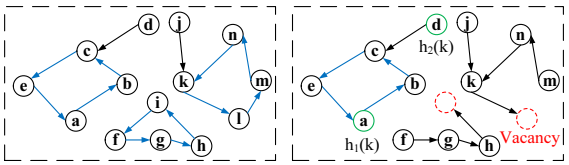
[1] https://github.com/syy804123097/SmartCuckoo.

it is an undirected graph in which each subgraph has no more edges than vertices. In a pseudoforest, two cycles composed of consecutive edges share no vertices with each other, and cannot be linked to each other by a path of consecutive edges.

In order to show the difference of actual and backup positions of items, we take into consideration the direction of kick-out operations. In a directed graph, each edge is directed from one of its endpoints to the other. Each bucket in the hash tables stores at most one item, and thus each vertex in a directed pseudoforest has an outdegree of at most one. If a subgraph contains a vertex whose outdegree is zero, it does not contain a cycle and the vertex corresponds to a vacant slot. Otherwise, it contains a cycle and any insertion into the subgraph will walk into an endless loop [31].

Definition 2 ***Maximal Directed Pseudoforest.*** *A maximal directed pseudoforest is a directed graph in which each vertex has an outdegree of exactly one.*

We name a subgraph whose number of vertices are equal to its number of edges a **maximal subgraph**. A maximal subgraph contains a cycle. Any subgraph in a maximal directed pseudoforest is a maximal subgraph. Figure 3(a) shows an example of a maximal directed pseudoforest. There are three maximal subgraphs in a maximal directed pseudoforest. In contrast, a non-maximal directed pseudoforest has at least one non-maximal subgraph, namely, has at least one vertex whose outdegree is zero. As illustrated in Figure 3(b), the non-maximal directed pseudoforest has three subgraphs, two of which do not have any cycles. It can be transformed to a maximal directed pseudoforest by connecting any vertex whose outdegree is zero (the dotted circles in Figure 3(b)) with any other vertex in the graph by adding a new edge.



(a) Maximal. (b) Non-Maximal.

Figure 3: The Directed Pseudoforest.

We consider the cuckoo graph as a directed pseudoforest. Each vertex of the pseudoforest corresponds to a bucket of the hash tables and each edge corresponds to an item between two candidate positions of the item. An inserted item hence produces an edge. According to the property, a maximal subgraph has no room to admit a new edge, which eventually causes an endless loop when the directed edges are traversed. Such an endless loop will not be encountered in a non-maximal subgraph, which does not contain a cycle.

3 The SmartCuckoo Design

As a cost-efficient variant of cuckoo hashing, SmartCuckoo maintains high lookup efficiency and improves the insertion performance by avoiding unnecessary kick-out operations. It classifies item insertions into three cases and leverages a directed pseudoforest to represent hashing relationship, which is used to track item placements for accurately predicting the occurrence of endless loops. Conventional cuckoo hashing chooses one of the candidate positions for an item's placement without considering whether it would walk into an endless loop. Our design increases insertion efficiency by tracking status of subgraphs to predict the insertion walk outcome. Hence, SmartCuckoo intelligently selects insertion positions for the item to be inserted. In addition, we also illustrate the execution of operations in SmartCuckoo, including item insertion and deletion.

3.1 The Directed Pseudoforest Subgraph

Inserted items in cuckoo hashing form a *cuckoo graph*. We represent the cuckoo graph as a directed pseudoforest, which can reveal the path, consisting of directed edges, of kick-out operations for insertion. Hence, the directed graph can be used to track and tell endless loops in advance to avoid them.

Successful item insertion depends on finding a vacant bucket for storage. To this end, one of candidate buckets of an item to be inserted must belong to a subgraph containing one vertex whose outdegree is zero, corresponding to a vacant slot. Hence, detecting vacancies in a subgraph is crucial in the insertion operation of cuckoo hashing. Knowing the path of a sequence of kick-out operations for an item's insertion before the insertion is carried out will help to identify and avoid an endless loop. In our design, we characterize the cuckoo hashing as a directed graph, in which a bucket is represented as a vertex and an item is represented as an edge between two candidate positions of an item. SmartCuckoo stores at most one item in each bucket, and each item has a unique backup position. Accordingly, each edge has a start point representing the actual storage position of the item and an end point representing the backup one. In the directed graph, each vertex corresponds to a bucket and each edge corresponds to an item. Because items stored in a hash table are always not more than the buckets, the number of vertices is not smaller than that of edges in the directed graph. Therefore, there is at most one cycle existing in a subgraph. Hence, according to the

property of the directed pseudoforest, the directed graph used to characterize item placements in SmartCuckoo is a directed pseudoforest.

Furthermore, we have the following observation. When inserting a new item into a non-maximal directed subgraph of a pseudoforest, it will be stored in one of its candidate buckets, and then one kicked-out item will be stored in the vacant bucket corresponding to the last vertex of a directed cuckoo path. If one attempts to insert the item into a maximal directed pseudoforest, an endless loop will inevitably occur. Each vertex in a directed pseudoforest has an outdegree of one, except those with an outdegree of zero representing vacant buckets located at the ends of the directed paths in the non-maximal subgraphs. In a maximal directed pseudoforest, each vertex has an outdegree of one and no vertex can be the destination on the path of kick-out operations to store the item for insertion. That is, an endless loop is encountered.

The observation inspires us to design a strategy on the selection of a path leading to a vacant position for item insertion. Vertices of ourdegree of zero, which represent vacant positions (buckets) in the directed pseudoforest, are produced by prior item insertions. To reach a vacant vertex in a directed pseudoforest for inserting an item, at least one of the item's candidate buckets must be in a subgraph containing a vacant position. Figure 3(b) illustrates the process of inserting Item *k*. Its two candidate positions are currently occupied by Items *a* and *d* (green vertices) and are in a subgraph without vacant positions. Its insertion would encounter an endless loop and fail, though there exist two vacancies (red vertices) in the pseudoforest. Because only non-maximal subgraphs contain vacant positions, the success of an insertion of an item relies on whether at least one of its candidate positions is in a non-maximal subgraph.

New item insertions can be classified into three cases, i.e., $\mathbf{v+2}$, $\mathbf{v+1}$, and $\mathbf{v+0}$. As each item is represented as an edge in the pseudoforest, different placements of the item will increase the graph's vertex count differently (by two, one, or zero).

3.2 Three Cases of Item Insertions

In the implementation of conventional cuckoo hashing, an insertion failure is not known until a kick-out path is tried and a threshold of steps on the path is reached without locating a vacant position. The lack of a priori knowledge in the traditional implementations often leads to walking into endless loops with substantial time and resources spent on fruitless tries. To obtain the knowledge on endless loops in SmartCuckoo, we classify item insertions according to the number of additional vertices added to the directed pseudoforest.

In a directed pseudoforest, each edge corresponds to an inserted item, and each vertex corresponds to a bucket. Hence, for each item to be inserted into the hash tables, the number of edges is incremented by one. However, the increase of vertex count (v) has three cases, namely, the cases of $v+0$, $v+1$, and $v+2$. In the last two cases, the new item can be successfully inserted, which will be explained. Here we first discuss the status of the directed pseudoforest in the case of $v+0$.

3.2.1 The Case of $v+0$

When inserting an item without increasing vertex count, two vertices corresponding to two candidate buckets of the item should have existed in the directed pseudoforest, which leads to five possible scenarios, as illustrated in Figure 4.

- Two candidate buckets of Item x_1, shown as blue buckets in Figure 4(a), exist in the same non-maximal directed subgraph A. Either bucket can be selected to have a successful insertion as the kick-out operations will always reach a vacant position in the subgraph. As shown in Figure 4(a), Item x_1 is directly inserted into Bucket $T_2[3]$ and creates a new edge from Bucket $T_2[3]$ to Bucket $T_1[0]$, which is the backup position of Item x_1. After the insertion of Item x_1, the original non-maximal directed subgraph A is transformed into a maximal directed subgraph A', which does not have a vacant position to admit a new item.

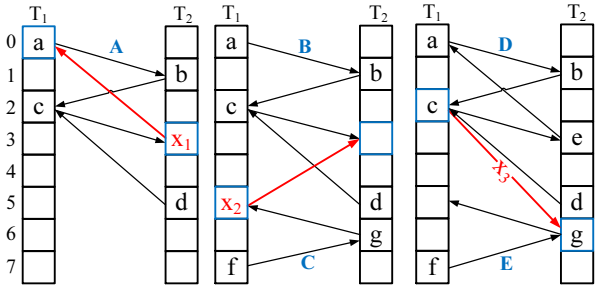
- Two candidate buckets of Item x_2 are in two different non-maximal directed subgraphs B and C, respectively, as shown in Figure 4(b). In this scenario, the insertion operation will also be a success, because each of two non-maximal directed subgraphs offers a vacant bucket. Item x_2 is located in Bucket $T_1[5]$ and constructs a new directed edge from Bucket $T_1[5]$ to Bucket $T_2[3]$ in the directed pseudoforest, which merges the two subgraphs, B and C, into a new non-maximal directly subgraph (BC) with one vacant vertex ($T_2[3]$).

- One candidate bucket of Item x_3 is in the non-maximal directed subgraph E and the other is in the maximal directed subgraph D, as shown in Figure 4(c). If the item enters the hash table from Bucket $T_1[2]$, an endless loop is encountered in the maximal directed subgraph D and unnecessary kick-out operations are carried out. However, if Item x_3 enters the hash table at Bucket $T_2[6]$, the item insertion will be a success after a number of kick-out operations (simply kicking out Item g to Bucket $T_1[5]$ in the example shown in Figure 5(a)).

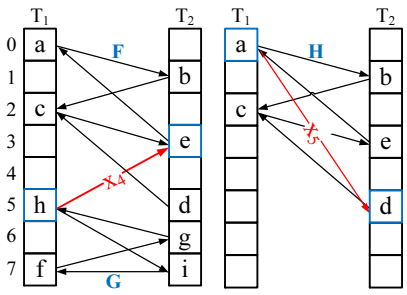
Accordingly, two subgraphs D and E are merged into a new maximal directed subgraph (DE), which does not have any vacant buckets.

- Two candidate buckets of Item x_4 are separated into two maximal directed subgraphs (F and G), as shown in Figure 4(d). Because there doesn't exist any vacant buckets in any of the subgraphs, the insertion of the new item (x_4) will always walk into an endless loop, illustrated in Figure 5(b). This is the worst scenario for an insertion in conventional cuckoo hashing implementations.

- Two candidate buckets of Item x_5 are in the same maximal directed subgraph (H), as shown in Figure 4(e). Similar to the previous scenario, the insertion will turn out to a failure after numerous kick-outs in an endless loop, as shown in Figure 5(c).



(a) One non-maximal. (b) Two non-maximal. (c) One maximal and one non-maximal.

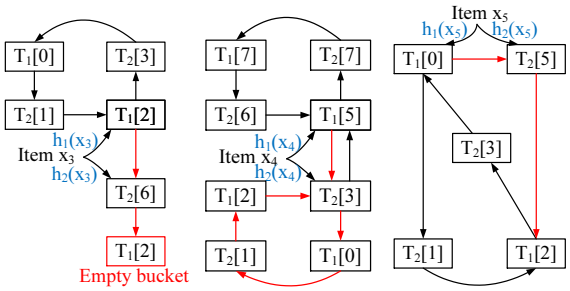


(d) Two maximal. (e) One maximal.

Figure 4: Five scenarios for Case $v+0$.

3.2.2 The Cases of $v+1$ and $v+2$

The $v+1$ represents the case where the number of vertices in the directed pseudoforest is increased by 1 after insertion of an item. As shown in Figure 6(a), in this case one of two candidate positions of Item x_6 corresponds to an existing vertex in the directed pseudoforest. The other will be a new vertex after the item's insertion. That is, this candidate bucket in the

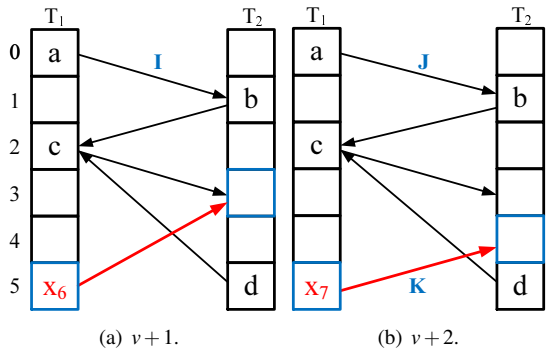


(a) One maximal and one non-maximal. (b) Two maximal. (c) One maximal.

Figure 5: Buckets that are accessed during kick-out operations.

hash table has not been represented by any vertices in the pseudoforest. Item x_6 is then placed in this position, and a new edge connecting the new vertex with the existing vertex is added into the subgraph I of the directed pseudoforest.

In the case of $v+2$, both candidate positions of an item to be inserted have not yet been represented by any vertices in the pseudoforest. Accordingly, they are unoccupied. The item can be inserted in any of the two available positions. Accordingly, two vertices, each corresponding to one of the positions, are added into the pseudoforest. Furthermore, an edge from the vertex corresponding to the position where the item is actually placed to the other corresponding to its backup position is also added. The two vertices and the new edge constitute a new subgraph (K), which is a non-maximal directed one. This case is illustrated in Figure 6(b), where the two vertices are Buckets $T_1[5]$ and $T_2[4]$, and the new edge is from Bucket $T_1[5]$ to Bucket $T_2[4]$ after Item x_7 is inserted at Bucket $T_1[5]$.



(a) $v+1$. (b) $v+2$.

Figure 6: The cases of $v+1$ and $v+2$.

3.3 Predetermination of An Endless Loop

According to Section 3.2, if we know in advance which case an item insertion belongs to, we can predetermine whether any of the item's candidate positions is on an endless loop. This is achieved by tracking status of subgraphs, which is either *maximal directed* or *non-maximal directed*. If a candidate position is in a maximal directed subgraph, it is on an endless loop. Otherwise, it is not on an endless loop.

During an item insertion operation, for each of its candidate positions, we need to find out which one or two subgraphs of a directed pseudoforest it belongs to. To this end, we apply the *Find* operation for a given candidate position to determine the subgraph it belongs to. In addition, if two candidate positions of an inserted item belong to two subgraphs, an edge will be introduced between the subgraphs and the two subgraphs need to be merged. To this end, a *Union* operation is required to merge them. To enable Find and Union operations in SmartCuckoo, we assign each subgraph a unique ID. Each member vertex of the subgraph records the ID in its corresponding bucket. When two subgraphs are merged into a new one, instead of exhaustively searching for member vertices of one or two of the original subgraphs on the hash tables to update their subgraph ID, we introduce trees of the IDs. In a tree for merged subgraphs, the IDs at the buckets representing the subgraphs before the merging are leaf nodes and the ID of the new subgraph is the parent. The new subgraph is likely to be merged again with another subgraph and has its parent. In the end, the ID at the root of the tree represents the subgraph merged from all the previous subgraphs.

In order to determine the status of a subgraph in the pseudoforest, we track its edge count and vertex count. A subgraph is a maximal directed one if its edge count is equal to its vertex count. In this case, the subgraph does not have any room to admit new edges. Otherwise, the edge count is smaller than vertex count, and the subgraph is a non-maximal directed one.

In summary, we can predetermine the outcome of an item insertion based on the statuses of related subgraphs in each of the three cases the insertion belongs to.

- **v + 2:** When the two candidate positions (a and b) of Item x have not yet been represented by any vertices in a directed pseudoforest, the insertion will create a new subgraph, which is non-maximal directed. Therefore, the new Item x can be successfully inserted. Moreover, the vertex count of the subgraph is 2, and the edge count is 1.

- **v + 1:** This case is detected after running $Find(a)$ and $Find(b)$ and finding out that one of the candidate positions corresponds to an existing vertex in a subgraph and the other has not yet been represented by any vertex. In this case, no matter which status the subgraph is on, the insertion will be a success due to the introduction of a new vertex. Both the vertex count and the edge count of the subgraph are increased by 1.

- **v + 0:** In this case, both candidate positions are vertices in subgraphs. To know the outcome of the insertion, we need to determine the status of the subgraphs. Only if at least one of the subgraphs is non-maximal, the insertion is a success. Otherwise, the insertion would fail after walking into an endless loop. The edge count of the corresponding subgraph is increased by 1.

3.4 Implementations of Operations

In the Section, we describe how two common hash table operations, namely insertion and deletion, are supported in SmartCuckoo, as its implementation of lookup operation is essentially the same as that in conventional cuckoo hashing.

3.4.1 Insertion

We use $B[*]$ to represent the item in the bucket. Algorithm 1 describes the steps involved in the insertion of Item x. First, we determine the case the insertion belongs to and increases corresponding vertex count (v), as described in Algorithm 2. The t value indicates one of the three cases ($v+2$, $v+1$, and $v+0$) for the insertion. In the cases of $v+1$ and $v+2$, Item x can be directly inserted, as described in Algorithm 3. If the insertion case is $v+0$, we use Algorithm 4 to determine which of the following five scenarios about corresponding subgraph(s) applies: (1) one non-maximal, (2) two non-maximal (Lines 4-6), (3) one non-maximal and one maximal (Lines 7-13), (4) two maximal, and (5) one maximal. SmartCuckoo avoids walking into a maximal directed subgraph. Due to no loops, SmartCuckoo is able to efficiently reduce the repetitions in one path, thus reducing insertion operation latency.

3.4.2 Deletion

An item can only be stored in one of the candidate positions of the hash tables. During the deletion operation, we only need to probe the candidate positions and, if found at one of the positions, remove it from the position (Lines 3-4). Deleting an item from the hash tables is equivalent to removal of an edge in the corresponding subgraph, which causes the subgraph to be separated into two subgraphs. We assign each

Algorithm 1 Insert(Item x)

```
1: a ← Hash1(x)
2: b ← Hash2(x) /*Two candidate positions of Item x*/
3: t ← Determine-v-add(a,b)
4: if t == v+2 then
5:     Assign a unique ID to the new subgraph
6:     Union(a,b)
7:     DirectInsert(x,a,b)
8:     Return Ture /*Finish the insertion*/
9: else if t == v+1 then
10:    Union(a,b)
11:    DirectInsert(x,a,b)
12:    Return True /*Finish the insertion*/
13: else
14:    InDirectInsert(x,a,b)
15: end if
```

Algorithm 2 Determine-v-add(Hash a, Hash b)

```
1: if neither a nor b have yet existed in the pseudoforest then
2:     Return v+2
3: else if both a and b have existed in the pseudoforest then
4:     Return v+0
5: else
6:     Return v+1
7: end if
```

Algorithm 3 DirectInsert(Item x, Hash a, Hash b)

```
1: /*a and b are two candidate positions of Item x*/
2: if B[a] is empty then
3:     B[a] ← x /*Insert Item x into the empty bucket*/
4: else
5:     B[b] ← x
6: end if
```

Algorithm 4 InDirectInsert(Item x, Hash a, Hash b)

```
1: /*Determine type of the corresponding subgraphs*/
2: if one non-maximal then
3:     Kick-out(x,B[a])
4: else if two non-maximal then
5:     Kick-out(x,B[a])
6:     Union(a,b)
7: else if one non-maximal and one maximal then
8:     if the subgraph containing a is non-maximal then
9:         Kick-out(x,B[a])
10:    else
11:        Kick-out(x,B[b])
12:    end if
13:    Union(a,b)
14: else
15:    Rehash()
16: end if
```

of the two subgraphs a new ID, and update the IDs of each member vertex of the two subgraph in their corresponding buckets (Lines 5-6). In addition, the vertex count and edge count of the two subgraphs are updated. Algorithm 5 describes how the pseudoforest is maintained in the deletion of Item x.

4 Performance Evaluation

4.1 Experimental Setup

The server used in our experiments is equipped with an Intel 2.8GHz 16-core CPU, 12GB DDR3 RAM with a peak bandwidth of 32GB/s, and a 500GB hard disk. The L1 and L2 caches of the CPU are 32KB and 256KB, respectively. We use three traces (RandomInteger [40], MacOS [3, 47], and DocWords [4]), and the YCSB benchmark [11] to run the SmartCuckoo prototype in the Linux kernel 2.6.18 to evaluate its performance. In addition, SmartCuckoo is implemented based on *CHS*.

RandomInteger: We used C++'s STL Mersenne Twister random integer generator [40] to generate items, which are in the full 32-bit unsigned integer range and follow a pseudo-random uniform distribution.

MacOS: The trace was collected on a Mac OS X Snow Leopard server [3,47]. We use fingerprints of files as keys to generate insertion requests. The fingerprints are obtained by applying the MD5 function on the file contents.

DocWords: This trace includes five text collections in the form of bag-of-words [4]. It contains nearly 80 million items in total. We take advantage of the combination of its *DocID* and *WordID* as keys of items to be inserted into hash tables.

We compare SmartCuckoo with CHS (*cuckoo hashing with a stash*) [29] as the *Baseline*, *libcuckoo* [36], and *BCHT* [15] schemes. Specifically, for *BCHT*, we implemented its main components, including four slots in each bucket. For *libcuckoo*, we use its open-source C++ implementation [1], which is optimized to serve write-heavy workloads.

4.2 Results and Analysis

We present evaluation results of SmartCuckoo and compare them with those from the state-of-the-art cuckoo hash tables in terms of insertion throughput, lookup throughput, and the throughput of mixed operations.

4.2.1 Insertion Throughput

Figure 7 shows the insertion throughputs of *Baseline*, *libcuckoo*, *BCHT*, and the proposed SmartCuckoo with the RandomInteger workload. With the increase of the load factor, we observe that SmartCuckoo significantly increases insertion throughput over *Baseline* by 25% to 75%, *libcuckoo* by 65% to 75%, and over *BCHT* by

Algorithm 5 Deletion(Item x)

```
1: a ← Hash1(x)
2: b ← Hash2(x) /*Two candidate positions of Item x*/
3: if x == B[a] or x == B[b] then
4:     Delete x from the corresponding position
5:     Assign two unique IDs to two new subgraphs respectively
6:     Update subgraph ID
7:     Update vertex and edge count
8:     Return True
9: else
10:    Return False
11: end if
```

40% to 50%. Conventional cuckoo hash tables, including *Baseline*, *libcuckoo*, and *BCHT*, essentially take a random walk to find a vacant bucket for inserting an item without a priori knowledge on the path, which leads to unnecessary operations and the extended response time. In particular, in addition to the impact of endless loops, *libcuckoo* suffers from frequent use of locking for consistent synchronization in its support of concurrent accesses. *BCHT* uses multi-slot buckets to mitigate the occurrence of endless loops. However, it requires a search in at least one candidate bucket for an available slot to carry out an insertion, which compromises its insertion throughput. In contrast, SmartCuckoo classifies item insertions into three cases to predetermine outcome of an insertion, so that an insertion failure can be known without actually performing any kick-out operations to significantly save insertion time. This performance advantage is particularly large with a hash table of a high load factor.

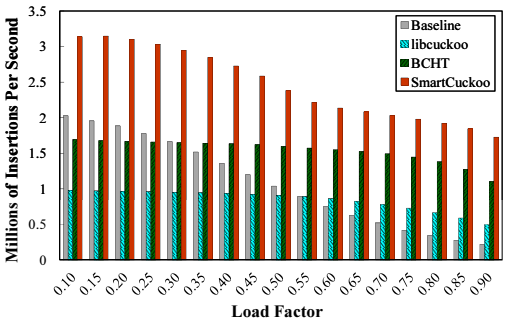


Figure 7: Insertion throughputs with RandomInteger.

Figure 8 shows insertion throughputs of the various cuckoo hash tables with the MacOS workload. Compared with conventional hash tables, SmartCuckoo obtains an average of 90% throughput improvement over *Baseline* at a load factor of 0.9, and 75% over *libcuckoo*, as well as 25% over *BCHT*.

Figure 9 illustrates the insertion throughputs with the

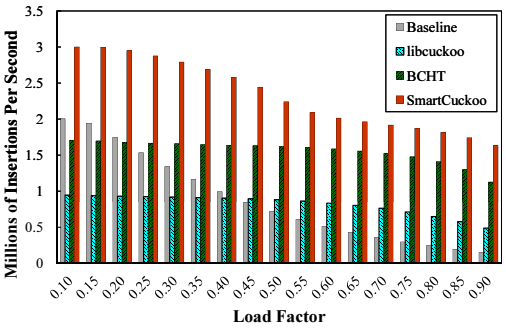


Figure 8: Insertion throughput with MacOS.

DocWords workload. With the increase of the load factor, SmartCuckoo increases insertion throughput over *Baseline* by 33% to 77%, *libcuckoo* by 60% to 75%, and over *BCHT* by 35% to 44%.

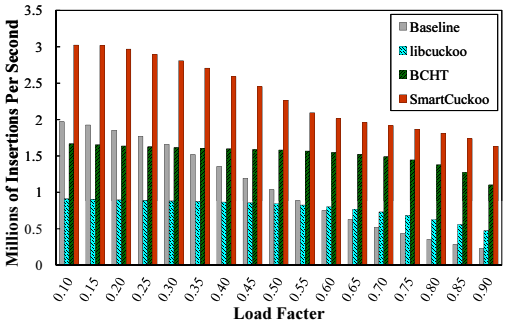


Figure 9: Insertion throughput with DocWords.

4.2.2 Lookup Throughput

In the evaluation of lookup performance of the four hash tables (*Baseline*, *libcuckoo*, *BCHT*, and SmartCuckoo), we generate the workload of all-lookup queries from each of the real-world traces. First, we extract lookup queries from a trace and use the remaining insertion and deletion queries in the trace to populate a hash table. Second, we selectively issue lookup queries, in the order of their appearance in the original trace, to the hash table. For a workload of lookup queries for only existent keys, we skip those for non-existent keys. For a workload of lookup queries for only non-existent keys, we skip those for existent keys. Each workload contains one million queries.

We examine the lookup throughputs of *Baseline*, *libcuckoo*, *BCHT*, and SmartCuckoo with the RandomInteger workload, which are shown in Figure 10. We observe that SmartCuckoo and *Baseline* achieve almost the same lookup throughput due to similar implementation of lookup operation. When all of the keys in the lookup queries are existent in the table,

SmartCuckoo improves the lookup throughputs by 30% and 5% over those of *libcuckoo* and *BCHT*, respectively. When none of the keys are in the table, all candidate positions (slots in *BCHT*) for a key have to be accessed. In particular, *BCHT* searches eight slots (four slots per bucket in the experiment setup) in two candidate buckets for each key, resulting in the reduced throughput.

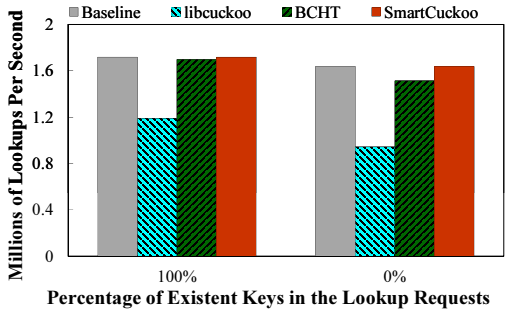


Figure 10: Lookup throughput with RandomInteger.

Figure 11 shows the lookup throughputs with the MacOS workload. Similarly, the throughput of SmartCuckoo is about 30% and 6% higher than that of *libcuckoo* and *BCHT*, respectively, with lookups of only existent keys. If all of the keys are non-existent, the improvements become 45% and 10%, respectively. Figure 12 shows the lookup throughputs with the DocWords workload, revealing similar performance trend.

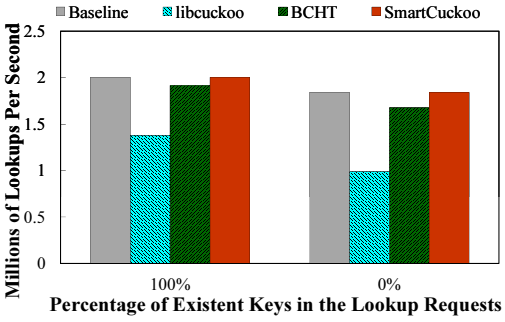


Figure 11: Lookup throughput with MacOS.

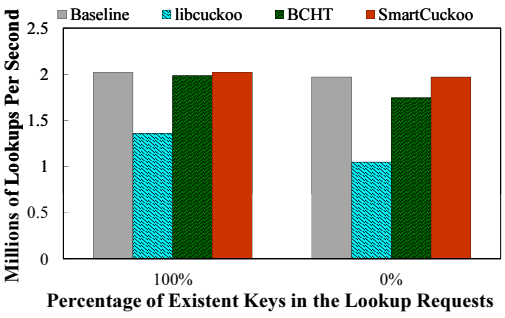


Figure 12: Lookup throughput with DocWords.

4.2.3 Throughput of Workload with Mixed Queries

We use YCSB [11] to generate five workloads, each with ten million key-value pairs, following the zipf distribution. Each key in the workloads is 16 bytes and each value is 32 bytes. The distributions of different types of queries in each workload are shown in Table 1.

Table 1: Distributions of different types of queries in each workload.

Workload	Insert	Lookup	Update
YCSB-1	100	0	0
YCSB-2	75	25	0
YCSB-3	50	50	0
YCSB-4	25	75	0
YCSB-5	0	95	5

Figure 13 shows the throughputs of SmartCuckoo and other hash tables in comparison for running each of the YCSB workloads. With the decrease of the percentage of insertions in the workloads, throughputs of all the cuckoo hash tables increase due to expensive kick-out operations during execution of insertion operations. With the workloads containing insert queries (the first four in Table 1), SmartCuckoo consistently produces higher throughput than the other three cuckoo hash tables, specifically by 25% to 70% than *Baseline*, by 25% to 55% than *libcuckoo*, and by 10% to 50% than *BCHT*. SmartCuckoo takes advantage of its ability of predetermining the occurrence of endless loops to avoid a potentially large number of step-by-step kick-out operations. The fifth workload does not have any insert queries. Instead, it does have a small percentage of update query, whose cost is similar to that of lookup if updating existent keys and is equivalent to that of insert if updating non-existent keys. Because most queries are for existent keys, SmartCuckoo and *Baseline* achieve nearly the same performance due to their similar implementation of lookup operation.

For each of the YCSB workloads that has 10 million key-value pairs with 16B keys and 32B values, the minimal space for holding the data in the cuckoo hash table is $(16+32)*10M = 458MB$. Any space additional to this minimal requirement to hold auxiliary data structure for higher performance is considered as the hash table's space overhead. With its use of a lightweight pseudoforest, SmartCuckoo has a space overhead of about 20% of the minimal requirement in the YCSB workloads. This is in line with that of the other three hash tables (*Baseline*, *libcuckoo*, and *BCHT*).

5 Related Work

Cuckoo Hashing Structures. SmartCuckoo is a variant of the cuckoo hashing, which supports fast and cost-

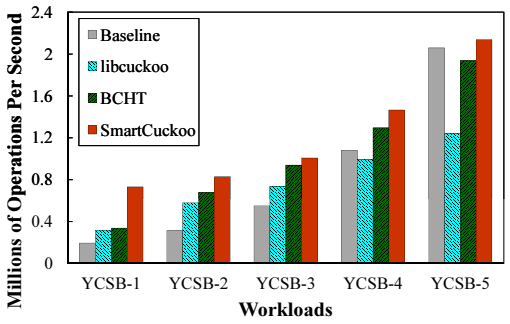


Figure 13: Throughput of mixed operations with YCSB.

efficient lookup operation. Cuckoo hashing [42, 43] is an open-addressing hashing scheme that provides each item with multiple candidate positions in the hash table. Studies of cuckoo hashing via the graph theory provide insightful understanding of cuckoo hashing's advantages and limitations [14, 32]. Cuckoo Graph is proposed to describe the hashing relationship of cuckoo hashing [32]. Cuckoo filter [17] uses the cuckoo hash table to enhance Counting Bloom filter [18] for supporting insertion and deletion operations with improved performance and space efficiency. Horton table [8] is an enhanced bucketized cuckoo hash table to reduce the number of CPU cache lines that are accessed in each lookup. In contrast, we investigate the characteristics of the directed cuckoo graph describing the kick-out behaviors. The proposed SmartCuckoo leverages the directed pseudoforest, a concept in graph theory, to track item placements in the hash table for predetermining occurrence of endless loops.

Content-based Search. NEST [24] uses cuckoo hashing to address the load imbalance issue in the traditional locality-sensitive hashing (LSH) and to support approximate queries. HCTrie [41] is a multi-dimensional structure for file search using scientific metadata in file systems, which supports a large number of dimensions. MinCounter [46] allocates a counter for each bucket to track kick-out times at the bucket, which mitigates the occurrence of endless loops during data insertion by selecting less used kick-out routes. SmartCuckoo aims at avoiding unnecessary kick-out operations due to endless loops in item insertion.

Searchable File Systems. Many efforts have been made to improving performance of large-scale searchable storage systems. Spyglass [34] is a file metadata search system, based on hierarchical partitioning of namespace organization, for high performance and scalability. Smartstore [23] reorganizes file metadata based on file semantic information for next-generation file systems. It provides efficient and scalable complex queries and enhances system scalability and functionality. Glance [25] is a just-in-time sampling-based system to provide accurate answers for aggregate and top-k queries without prior knowledge. Ceph [39, 53] uses dynamic subtree partitioning to support filename-based query as well as to avoid metadata-access hot spots. SmartCuckoo provides fast query services for cloud storage systems.

6 Conclusion and Future Work

Fast and cost-efficient query services are important to cloud storage systems. Due to the salient feature of open addressing, cuckoo hashing supports fast queries. However, it suffers from the problem of potential endless loops during item insertion. We propose a novel cost-efficient hashing scheme, named SmartCuckoo, for tracking item placements in the hash table. By representing the hashing relationship as a directed pseudoforest, SmartCuckoo can accurately predetermine the status of cuckoo operations and endless loops. We further avoid walking into an endless loop, which always belongs to a maximal subgraph in the pseudoforest. We use three real-world traces, i.e., RandomInteger, MacOS, and DocWords, and the YCSB benchmark to evaluate the performance of SmartCuckoo. Extensive experimental results demonstrate the advantages of SmartCuckoo over state-of-the-work schemes, including cuckoo hashing with a stash, *libcuckoo*, and *BCHT*.

SmartCuckoo currently addresses the issue of endless loop for cuckoo hash tables using two hash functions. It is well-recognized that using more than two hash functions would significantly increase operation complexity and is thus less used [13, 56]. A general and well-known approach is to reduce the number of hash functions to two using techniques such as double hashing [30]. As a future work, we plan to apply the approach of SmartCuckoo on hash tables using more than two hash functions. In addition, we will also study the use of SmartCuckoo in cuckoo hash tables with multiple slots in each bucket.

Acknowledgments

This work was supported by National Key Research and Development Program of China under Grant 2016YFB1000202 and State Key Laboratory of Computer Architecture under Grant CARCH201505. Song Jiang was supported by US National Science Foundation under CNS 1527076. The authors are grateful to anonymous reviewers and our shepherd, Rong Chen, for their constructive feedbacks and suggestions.

References

[1] Libcuckoo library. *https://github.com/efficient/libcuckoo*.

[2] Symantec. 2010 State of the Data Center Global Data. *http://www.symantec.com/content/en/us/about/media /pdfs/Symantec_DataCenter10_Report_Global.pdf* (Jan. 2010).

[3] Traces and Snapshots Public Archive. *http://tracer.filesystems.org* (July 2014).

[4] Bag-of-words data set. *http://archive.ics.uci.edu/ml/datasets /Bag+of+Words* (Mar. 2008).

[5] ÀLVAREZ, C., BLESA, M., AND SERNA, M. Universal Stability of Undirected Graphs in the Adversarial Queueing Model. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures* (2002), ACM, pp. 183–197.

[6] ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A. D., KATZ, R., KONWINSKI, A., LEE, G., PATTERSON, D., RABKIN, A., STOICA, I., ET AL. A View of Cloud Computing. *Communications of the ACM 53*, 4 (2010), 50–58.

[7] BELL, G., HEY, T., AND SZALAY, A. Beyond the Data Deluge. *Science 323*, 5919 (2009), 1297–1298.

[8] BRESLOW, A. D., ZHANG, D. P., GREATHOUSE, J. L., JAYASENA, N., TULLSEN, D. M., XU, L., CAVAZOS, J., ALVAREZ, M. A., MORALES, J. A., AGUILERA, P., ET AL. Horton Tables: Fast Hash Tables for In-Memory Data-Intensive Computing. USENIX Association, pp. 281–294.

[9] BÜTTCHER, S., AND CLARKE, C. L. A Security Model for Full-Text File System Search in Multi-User Environments. In *Proc. FAST* (2005).

[10] BYKOV, S., GELLER, A., KLIOT, G., LARUS, J. R., PANDYA, R., AND THELIN, J. Orleans: Cloud Computing for Everyone. In *Proc. SOCC* (2011), ACM, p. 16.

[11] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking Cloud Serving Systems with YCSB. In *Proc. SoCC* (2010), ACM, pp. 143–154.

[12] DAVID, T., GUERRAOUI, R., AND TRIGONAKIS, V. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. *ACM SIGARCH Computer Architecture News 43*, 1 (2015), 631–644.

[13] DEBNATH, B. K., SENGUPTA, S., AND LI, J. ChunkStash: Speeding up Inline Storage Deduplication using Flash Memory. In *Proc. USENIX ATC* (2010).

[14] DEVROYE, L., AND MORIN, P. Cuckoo hashing: Further analysis. *Information Processing Letters 86*, 4 (2003), 215–219.

[15] ERLINGSSON, U., MANASSE, M., AND MCSHERRY, F. A Cool and Practical Alternative to Traditional Hash Tables. In *Proc. WDAS* (2006).

[16] FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proc. NSDI* (2013).

[17] FAN, B., ANDERSEN, D. G., KAMINSKY, M., AND MITZENMACHER, M. D. Cuckoo Filter: Practically Better Than Bloom. In *Proc. CoNext* (2014), ACM, pp. 75–88.

[18] FAN, L., CAO, P., ALMEIDA, J., AND BRODER, A. Z. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking (TON) 8*, 3 (2000), 281–293.

[19] FRIEZE, A., MELSTED, P., AND MITZENMACHER, M. An Analysis of Random-Walk Cuckoo Hashing. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*. Springer, 2009, pp. 490–503.

[20] GABOW, H. N., AND WESTERMANN, H. H. Forests, frames, and games: Algorithms for matroid sums and applications. *Algorithmica 7*, 1 (1992), 465–497.

[21] GANTZ, J., AND REINSEL, D. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the Future 2007* (2012), 1–16.

[22] HSU, C.-H., ZHANG, Y., LAURENZANO, M. A., MEISNER, D., WENISCH, T., MARS, J., TANG, L., AND DRESLINSKI, R. G. Adrenaline: Pinpointing and Reining in Tail Queries with Quick Voltage Boosting. In *Proc. HPCA* (2015), IEEE, pp. 271–282.

[23] HUA, Y., JIANG, H., ZHU, Y., FENG, D., AND TIAN, L. Smartstore: A New Metadata Organization Paradigm with Semantic-Awareness for Next-Generation File Systems. In *Proc. SC* (2009), ACM.

[24] HUA, Y., XIAO, B., AND LIU, X. Nest: Locality-aware Approximate Query Service for Cloud Computing. In *Proc. INFOCOM* (2013), IEEE, pp. 1303–1311.

[25] HUANG, H. H., ZHANG, N., WANG, W., DAS, G., AND SZALAY, A. S. Just-in-Time Analytics on Large File Systems. *IEEE Transactions on Computers 61*, 11 (2012), 1651–1664.

[26] HUSTON, L., SUKTHANKAR, R., WICKREMESINGHE, R., SATYANARAYANAN, M., GANGER, G. R., RIEDEL, E., AND AILAMAKI, A. Diamond: A Storage Architecture for Early Discard in Interactive Search. In *Proc. FAST* (2004), pp. 73–86.

[27] JANAPA REDDI, V., LEE, B. C., CHILIMBI, T., AND VAID, K. Web Search Using Mobile Cores: Quantifying and Mitigating the Price of Efficiency. In *Proc. ISCA* (2010), pp. 314–325.

[28] KIRSCH, A., AND MITZENMACHER, M. The Power of Qne Move: Hashing Schemes for Hardware. *IEEE/ACM Transactions on Networking 18*, 6 (2010), 1752–1765.

[29] KIRSCH, A., MITZENMACHER, M., AND WIEDER, U. More Robust Hashing: Cuckoo Hashing with a Stash. *SIAM Journal on Computing 39*, 4 (2009), 1543–1561.

[30] KNUTH, D. E. *The Art of Computer Programming: Sorting and Searching*, vol. 3. Pearson Education, 1998.

[31] KRUSKAL, C. P., RUDOLPH, L., AND SNIR, M. Efficient parallel algorithms for graph problems. *Algorithmica 5*, 1 (1990), 43–64.

[32] KUTZELNIGG, R. Bipartite Random Graphs and Cuckoo Hashing. In *Fourth Colloquium on Mathematics and Computer Science Algorithms, Trees, Combinatorics and Probabilities* (2006), Discrete Mathematics and Theoretical Computer Science, pp. 403–406.

[33] LAM, H., LIU, Z., MITZENMACHER, M., SUN, X., AND WANG, Y. Information Dissemination via Random Walks in d-Dimensional Space. In *Proc. SODA* (2012), SIAM, pp. 1612–1622.

[34] LEUNG, A. W., SHAO, M., BISSON, T., PASUPATHY, S., AND MILLER, E. L. Spyglass: Fast, Scalable Metadata Search for Large-Scale Storage Systems. In *Proc. FAST* (2009), pp. 153–166.

[35] LI, Q., HUA, Y., HE, W., FENG, D., NIE, Z., AND SUN, Y. Necklace: An Efficient Cuckoo Hashing Scheme for Cloud Storage Services. In *Proc. IWQoS* (2014), IEEE, pp. 153–158.

[36] LI, X., ANDERSEN, D. G., KAMINSKY, M., AND FREEDMAN, M. J. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proc. EuroSys* (2014), ACM.

[37] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZIS, G., AND CAMBLE, P. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *Proc. FAST* (2009), pp. 111–123.

[38] LIU, L., XU, L., WU, Y., YANG, G., AND GANGER, G. R. SmartScan: Efficient Metadata Crawl for Storage Management Metadata Querying in Large File Systems. *Parallel Data Laboratory* (2010), 1–17.

[39] MALTZAHN, C., MOLINA-ESTOLANO, E., KHURANA, A., NELSON, A. J., BRANDT, S. A., AND WEIL, S. Ceph as a scalable alternative to the Hadoop Distributed File System. *login: The USENIX Magazine 35*, 4 (2010), 38–49.

[40] MATSUMOTO, M., AND NISHIMURA, T. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 8*, 1 (1998), 3–30.

[41] OHARA, Y. HCTrie: A Structure for Indexing Hundreds of Dimensions for Use in File Systems Search. In *Proc. MSST* (2013), IEEE, pp. 1–5.

[42] PAGH, R., AND RODLER, F. Cuckoo hashing. In *Proc. ESA* (2001), Springer, pp. 121–133.

[43] PAGH, R., AND RODLER, F. F. Cuckoo hashing. *Journal of Algorithms 51*, 2 (2004), 122–144.

[44] POLYCHRONIOU, O., RAGHAVAN, A., AND ROSS, K. A. Rethinking SIMD Vectorization for In-Memory Databases. In *Proc. SIGMOD* (2015), ACM, pp. 1493–1508.

[45] ROSS, K. A. Efficient Hash Probes on Modern Processors. In *Proc. ICDE* (2007), IEEE, pp. 1297–1301.

[46] SUN, Y., HUA, Y., FENG, D., YANG, L., ZUO, P., AND CAO, S. MinCounter: An Efficient Cuckoo Hashing Scheme for Cloud Storage Systems. In *Proc. MSST* (2015), IEEE.

[47] TARASOV, V., MUDRANKIT, A., BUIK, W., SHILANE, P., KUENNING, G., AND ZADOK, E. Generating Realistic Datasets for Deduplication Analysis. In *Proc. USENIX ATC* (2012), USENIX Association, pp. 261–272.

[48] TAYLOR, T., COULL, S. E., MONROSE, F., AND MCHUGH, J. Toward Efficient Querying of Compressed Network Payloads. In *Proc. USENIX ATC* (2012), USENIX Association, pp. 113–124.

[49] TURNER, V., GANTZ, J. F., REINSEL, D., AND MINTON, S. The digital universe of opportunities: Rich data and the increasing value of the internet of things. *Framingham (MA): IDC* (2014).

[50] VAMANAN, B., SOHAIL, H. B., HASAN, J., AND VIJAYKUMAR, T. TimeTrader: Exploiting Latency Tail to Save Datacenter Energy for Online Search. In *Proc. MICRO* (2015), ACM, pp. 585–597.

[51] WANG, C., REN, K., YU, S., AND URS, K. M. R. Achieving Usable and Privacy-assured Similarity Search over Outsourced Cloud Data. In *Proc. INFOCOM* (2012), IEEE, pp. 451–459.

[52] WANG, K., XU, G., SU, Z., AND LIU, Y. D. GraphQ: Graph Query Processing with Abstraction Refinement-Scalable and Programmable Analytics over Very Large Graphs on a Single PC. In *Proc. USENIX ATC* (2015), USENIX Association, pp. 387–401.

[53] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D., AND MALTZAHN, C. Ceph: A Scalable, High-Performance Distributed File System. In *Proc. OSDI* (2006), USENIX Association, pp. 307–320.

[54] WU, S., LI, F., MEHROTRA, S., AND OOI, B. C. Query Optimization for Massively Parallel Data Processing. In *Proc. SOCC* (2011), ACM.

[55] 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.

[56] ZUO, P., AND HUA, Y. A Write-friendly Hashing Scheme for Non-volatile Memory Systems. In *Proc. MSST* (2017).

Repair Pipelining for Erasure-Coded Storage

Runhui Li, Xiaolu Li, Patrick P. C. Lee, Qun Huang
The Chinese University of Hong Kong
lrhdiy@gmail.com, {lixl,pclee}@cse.cuhk.edu.hk, huangqundl@gmail.com

Abstract

We propose *repair pipelining*, a technique that speeds up the repair performance in general erasure-coded storage. By pipelining the repair of failed data in small-size units across storage nodes, repair pipelining reduces the repair time to approximately the same as the normal read time to the same amount of data in homogeneous environments. We further extend repair pipelining for heterogeneous environments. We implement a repair pipelining prototype called ECPipe and integrate it as a middleware system into two open-source distributed storage systems HDFS and QFS. Experiments on a local testbed and Amazon EC2 show that repair pipelining significantly improves the performance of both degraded reads and full-node recovery over existing repair techniques.

1 Introduction

Distributed storage systems rely on data redundancy to provide fault tolerance, so as to maintain availability and durability. *Replication*, which is traditionally used by production systems [4, 11], provides the simplest form of redundancy by keeping identical copies of data in different storage nodes. However, the raw storage cost of replication is overwhelming, especially with the massive scale of data we face today. *Erasure coding* provides a low-cost redundancy alternative that incurs significantly lower storage overhead than replication at the same fault tolerance level [39]. In a nutshell, erasure coding transforms fixed-size units, called *blocks*, of original data into a set of coded blocks, such that any subset of a sufficient number of available coded blocks can reconstruct all original data. Today's distributed storage systems adopt erasure coding to protect data against failures in clustered [10, 15, 29] or geo-distributed environments [21, 33], and reportedly save PBs of storage [15, 21].

Although achieving storage efficiency, erasure coding has a drawback of incurring high repair penalty. Specifically, the repair of a single failed coded block (either lost or unavailable) needs to read multiple available coded blocks for reconstruction; in other words, it reads more available data than the actual amount of failed data. This is in contrast to replication, whose repair can be simply done by reading another replica that is of the same size as the failed block. The excessive data not only increases the read time to failed data as opposed to normal reads, but also consumes bandwidth resources that could otherwise be made available for other foreground jobs [29]. Thus, erasure coding in practice is mainly used for storing less frequently read (i.e., warm/cold) data that needs long-term persistence [2, 15, 21], while frequently read (i.e., hot) data remains replicated for efficient access. To mitigate the repair penalty of erasure coding, prior studies either propose new erasure codes that reduce the amount of repair traffic (e.g., [8, 15, 17, 25, 28, 30, 34]), or design fast repair approaches for existing erasure codes (e.g., lazy repair [3, 37] or parallel partial repair (PPR) [20]). While the repair time is effectively reduced, it remains higher than the normal read time in general. In view of this, we pose the following question: *Can we further reduce the repair time of erasure coding to almost the same as the normal read time?* This creates opportunity for applying erasure coding to hot data for high storage efficiency, while preserving read performance.

We present a new technique called *repair pipelining* to speed up the repair performance in general erasure-coded storage. Its main idea is to pipeline the repair of a coded block in small-size units across storage nodes (analogous to wormhole routing [22]), so as to distribute repair traffic and fully utilize bandwidth resources across storage nodes. Contrary to the conventional wisdom that the repair of erasure coding is a slow operation, repair pipelining can reduce the repair time of a failed coded block to approximately the same as the read time of a normal coded block, regardless of coding parameters, in homogeneous environments (i.e., link bandwidths are identical). It is also general to support various practical erasure codes that are adopted by today's production systems, including classical Reed-Solomon codes [32] and recent Local Reconstruction Codes [15]. To summarize, we make the following contributions.

- We design repair pipelining to address two types of repair operations: degraded reads and full-node recovery. We show that repair pipelining achieves $O(1)$ repair time in homogeneous environments.
- We extend repair pipelining to address heterogeneous environments (i.e., link bandwidths are different). We present two variants of repair pipelining. The first one allows parallel reads of reconstructed data when the bandwidth between the storage system and the node that issues repair is limited, while the second one finds

an optimal repair path across storage nodes such that the repair time is minimized.

- We implement a repair pipelining prototype called ECPipe, which runs as a middleware layer atop an existing storage system and performs repair operations on behalf of the storage system. As a proof of concept, we integrate ECPipe into two widely adopted open-source distributed storage systems HDFS [36] and QFS [24]. Both integrations only make minor changes (with no more than 200 lines of code) to the code base of each storage system.
- We evaluate repair pipelining on a local cluster and two geo-distributed Amazon EC2 clusters (one in North America and one in Asia). We compare it with two existing repair approaches: conventional repair that is used by classical Reed-Solomon codes [32] and achieves $O(k)$ repair time, and the recently proposed PPR [20] that achieves $O(\log k)$ repair time by parallelizing partial repair operations in a hierarchical manner (§2.2). Our experiments show that in many cases, repair pipelining reduces the single-block repair time by around 90% and 80% compared to conventional repair and PPR, respectively. It also improves repair performance in HDFS and QFS deployments.

2 Background and Motivation

2.1 Basics

We consider a distributed storage system (e.g., GFS [11], HDFS [36], and Azure [4]) that manages large-scale datasets and stores files as fixed-size *blocks*, which form the basic read/write units. The block size is often large, ranging from 64 MiB [11] to 256 MiB [30], to mitigate I/O overhead. Erasure coding is applied to a collection of blocks. Specifically, an erasure code is typically configured with two integer parameters (n, k), where $k < n$. An (n, k) code divides blocks into groups of k. For every k (uncoded) blocks, it encodes them to form n coded blocks, such that any k out of n coded blocks can be decoded to the original k uncoded blocks. The set of n coded blocks is called a *stripe*. A large-scale storage system stores data of multiple stripes, all of which are independently encoded. The n coded blocks of each stripe are distributed across n distinct nodes to tolerate any $n-k$ node failures. Most practical erasure codes are *systematic*, such that k of n coded blocks are identical to the original uncoded blocks and hence can be directly accessed without decoding. Nevertheless, our design treats both uncoded and coded blocks the same, so we simply refer to them as "blocks".

Many erasure code constructions have been proposed in the literature (see survey [26] and §7). Among all erasure codes, Reed-Solomon (RS) codes [32] are the most popular erasure codes that are widely deployed in production [10, 24, 29]. Note that RS codes achieve the minimum storage redundancy among any (n, k) codes, and are said to be *maximum distance separable (MDS)*. Some erasure codes used in production, such as locally repairable codes [15,34], introduce slightly higher redundancy than RS codes for better repair performance.

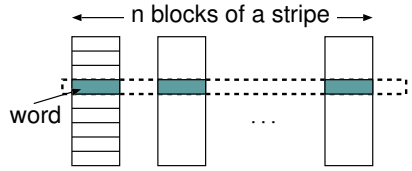


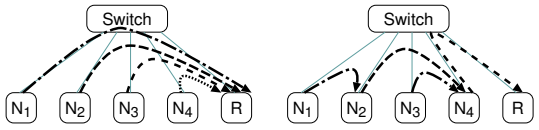
Figure 1: In erasure coding, blocks are partitioned into words, such that words at the same offset of each block of a stripe are encoded together.

Practical erasure codes, including RS codes and locally repairable codes, satisfy *linearity*. Specifically, for each stripe of an (n, k) code, let $\{B_1, B_2, \cdots, B_k\}$ denote any k blocks of a stripe. Any block in the same stripe, say B^*, can be computed from a linear combination of the k blocks as $B^* = \sum_{i=1}^{k} a_i B_i$, where a_i's $(1 \leq i \leq k)$ are decoding coefficients specified by a given erasure code. All additions and multiplications are based on Galois Field arithmetic over w-bit units called *words*; in particular, an addition is equivalent to bitwise XOR. Note that the additions of $a_i B_i$'s are associative. Some constraints may be applied; for example, RS codes require $n \leq 2^w + 1$ [27]. Each block is partitioned into multiple w-bit words, such that the words at the same offset of each block of a stripe are encoded together, as shown in Figure 1.

2.2 Repair

In this paper, repair in erasure-coded storage can refer to one of the following: (i) *full-node recovery* for restoring lost blocks (e.g., due to disk crashes, sector errors, etc.), or (ii) *degraded reads* to temporarily unavailable blocks (e.g., due to power outages, network disconnection, system maintenance, etc.) or lost blocks that are yet recovered. Each failed block (either lost or unavailable) is reconstructed on a destination termed *requestor*, which can be a new node that replaces a failed node, or a client that issues degraded reads. Note that there may be one or multiple requestors when multiple failed blocks are reconstructed.

Erasure coding triggers more repair traffic than the size of failed data to be reconstructed. For example, for (n, k) RS codes, repairing a failed block reads k available blocks of the same stripe from other nodes (i.e., k times the block size). Some repair-friendly erasure codes (e.g., [8, 15, 17, 25, 28, 30, 34]) are designed to reduce repair traffic, but the size of repair traffic per block remains larger than the size of a block. In distributed storage sys-



(a) Conventional repair (b) PPR

Figure 2: Examples of conventional repair and PPR.

tems, network bandwidth is often the most dominant factor in repair performance as extensively shown by previous work [8, 20, 37] (see further justifications in §2.3). Thus, the amplification of repair traffic implies the congestion at the downlink of the requestor, thereby increasing the overall repair time.

To understand the repair penalty of erasure coding, we use RS codes as an example and call this repair approach *conventional repair*. Suppose that a requestor R wants to repair a failed block B^*. It can be done by reading k available blocks from any k working nodes, called *helpers*. Without loss of generality, let R contact k helper nodes $N_1, N_2, \cdots, N_k$, which store available blocks B_1, $B_2, \cdots, B_k$, respectively. To make our discussion clear, we divide the repair process into *timeslots*, such that only one block can be transmitted across a network link in each timeslot. Figure 2(a) shows the conventional repair for $k = 4$. Since R needs to retrieve the k blocks B_1, B_2, $\cdots, B_k$, all k transmissions must traverse the downlink of R. Overall, the repair takes four timeslots.

The drawback of conventional repair is that the bandwidth usage distribution is highly skewed: the downlink of the requestor is highly congested, while the links among helpers are not fully utilized. PPR [20] builds on the linearity and addition associativity of erasure coding by decomposing a repair operation into multiple partial operations that are distributed across all helpers. This distributes bandwidth usage across the links of helpers. Figure 2(b) shows how PPR repairs B^* for $k = 4$. In the first timeslot, N_2 and N_4 receive blocks a_1B_1 and a_3B_3 from N_1 and N_3, respectively. Since the transmissions use different links, they can be done simultaneously in a single timeslot. In the second timeslot, N_2 combines the received a_1B_1 and its locally stored block B_2 to obtain $a_1B_1 + a_2B_2$ and sends it to N_4. In the third timeslot, N_4 combines all received blocks and its own block B_4 to obtain $a_1B_1 + a_2B_2 + a_3B_3 + a_4B_4$, and sends it to R. This hierarchical approach reduces the overall repair time to only three timeslots. In general, PPR needs $\lceil \log_2(k+1) \rceil$ timeslots to repair a failed block.

2.3 Motivation

Although PPR reduces repair time, the bandwidth usage distribution remains not fully balanced; for example, the downlink of N_4 in Figure 2(b) still carries more repair traffic than other links. Thus, the repair time is still bottlenecked by the link with the most repair traffic. This motivates us to design a new repair scheme that can more efficiently utilize bandwidth resources, with the primary goal of minimizing repair time.

Minimizing repair time is critical to both availability and durability. In terms of availability, field studies show that transient failures (i.e., no data loss) account for over 90% of failure events [10]. Thus, most repairs are expected to be degraded reads rather than full-node recovery. Since degraded reads are issued when clients request unavailable data, achieving fast degraded reads not only improves availability but is also critical for meeting customer service-level agreements [15]. In terms of durability, minimizing repair time also minimizes the window of vulnerability before unrecoverable data loss occurs.

Our work targets distributed storage environments in which network bandwidth is the bottleneck. Although modern data centers scale to 10Gb/s or higher speeds, they are shared by a mix of application workloads. Thus, the network bandwidth available for repair tasks is often throttled [15, 37]. Also, the cross-rack links of modern data centers are oversubscribed [5], yet blocks are striped across racks to tolerate rack failures [10, 15, 30, 34]. Repair of failed blocks inevitably reads available blocks from other racks, and its performance becomes constrained by the limited cross-rack bandwidth.

3 Repair Pipelining

We present the design of repair pipelining for both degraded reads and full-node recovery.

3.1 Goals and Assumptions

Repair pipelining also exploits the linearity and addition associativity of erasure codes as in PPR [20], yet it parallelizes the repair across helpers in an inherently different way. It focuses on (i) eliminating bottlenecked links (i.e., no link transmits more traffic than others) and (ii) effectively utilizing bandwidth resources during repair (i.e., links should not be idle for most times), so as to ultimately achieve $O(1)$ repair time in homogeneous environments where all links have the same bandwidth. In addition, we show that repair pipelining can be extended for practical distributed environments with heterogeneous links (§4), which are not addressed by PPR.

Repair pipelining is designed for speeding up the repair of a single failed block per stripe, which accounts for the most repair scenarios in practice [15, 29] (e.g., over 98% of cases [29]). If a stripe has multiple failed blocks, we trigger a multi-failure repair, in which we resort to conventional repair (§2.2) by reading a sufficient number of available blocks. Optimizing single-block repair is also the main design goal of repair-friendly erasure codes [8, 15, 17, 25, 28, 30, 34]. In this paper, we study the single-block repair for one stripe and multiple stripes. The former occurs when a requestor issues a de-

graded read to an unavailable block, which are the majority (§2.3); the latter occurs when all lost data of a single failed node is recovered at one or multiple requestors in full-node recovery.

As in PPR, we do not design new repair-friendly erasure codes that minimize repair traffic; instead, each repair of a single failed block still reads k blocks, yet it spreads the repair traffic across all helpers to fully utilize bandwidth resources and reduce the overall repair time.

3.2 Degraded Reads

We first study how repair pipelining reconstructs a single block of a stripe at a requestor in a degraded read. We start with a naïve approach. Specifically, we arrange k helpers and the requestor as a linear path, i.e., $N_1 \rightarrow N_2 \rightarrow \cdots \rightarrow N_k \rightarrow R$. At a high level, to repair a lost block B^*, N_1 sends a_1B_1 to N_2. Then N_2 combines a_1B_1 with its own block B_2 and sends $a_1B_1 + a_2B_2$ to N_3. The process repeats, and finally, N_k sends R the combined result, which is B^*. The whole repair incurs k transmissions that span across k different links. Thus, there is no bottlenecked link. However, this naïve approach underutilizes bandwidth resources, since there is only one block-level transmission in each timeslot. The whole repair still takes k timeslots, same as the conventional repair (§2.2).

Thus, repair pipelining decomposes the repair of a block into the repair of a set of s small fixed-size units called *slices* $S_1, S_2, \cdots, S_s$. It pipelines the repair of each slice through the linear path, and each slice-level transmission over a link only takes $\frac{1}{s}$ timeslots. Figure 3 shows how repair pipelining works for $k = 4$ and $s = 6$.

A slice can have an arbitrarily small size, provided that Galois Field arithmetic can be performed (§2.1). For RS codes, the minimum size of a slice is a w-bit word; if $w = 8$, a word denotes a byte. On the other hand, practical distributed storage systems store data in large-size blocks, typically 64 MiB or even larger (§2.1). Since a coding unit (i.e., word) has a much smaller size than a read/write unit (i.e., block), we can parallelize a block-level repair operation into more fine-grained slice-level repair sub-operations. Having small-size slices can improve parallelism, but also increases the overhead of issuing many requests for transmitting slices over the network. We study the impact of the slice size in §6.

We analyze the time complexity of repair pipelining. Here, we neglect the overheads due to computation and disk I/O, which we assume cost less time than network transmission; in fact, they can also be executed in parallel with network transmission in actual implementation (§5). Each slice-level transmission over a link takes $\frac{1}{s}$ timeslots. The repair of each slice takes $\frac{k}{s}$ timeslots to traverse the linear path, and N_1 starts to transmit the last slice after $\frac{s-1}{s}$ timeslots. Thus, the whole repair time,

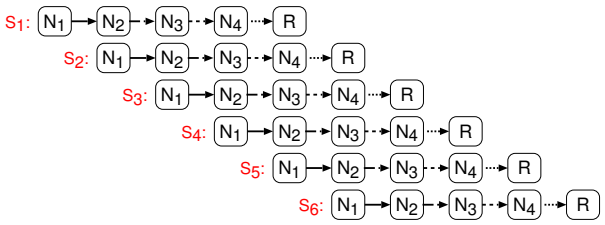


Figure 3: Repair pipelining with $k = 4$ and $s = 6$.

which is given by the total number of timeslots to transmit all slices through the linear path, is $\frac{s-1+k}{s} = 1+\frac{k-1}{s}$ timeslots. In practice, k is of moderate size to avoid large coding overhead [27] (e.g., $k = 12$ in Azure [15] and $k = 10$ in Facebook [29]), while s can be much larger (e.g., $s = 2{,}048$ for 32 KiB slices in a 64 MiB block). Thus, we have $1 + \frac{k-1}{s} \rightarrow 1$ as s is sufficiently large.

Repair pipelining connects multiple helpers as a chain, so its repair performance is degraded by the presence of poorly performed links/helpers (i.e., stragglers). We emphasize that any repair scheme of erasure coding faces the similar problem, as it needs available data from multiple helpers for data reconstruction; for example, the conventional repair of (n, k) MDS codes needs available data from k helpers. We address the straggler problem by taking into account heterogeneity and bypassing stragglers via helper selection (§4.2). Also, if any helper fails during an ongoing repair, the progress of repair pipelining will be stalled. In this case, we restart the whole repair process with a new set of available helpers and trigger a multi-failure repair (§3.1), since the repaired stripe now has multiple failed blocks; however, multi-failure repairs are rare in practice [15, 29].

3.3 Full-Node Recovery

We now study how repair pipelining addresses multi-stripe repair (one failed block per stripe) when recovering a full-node failure. As the stripes are independently encoded, we can parallelize the multiple single-stripe repair operations. However, since each repair involves a number of helpers, if one helper is chosen in many repair operations of different stripes, it will become overloaded and slow down the overall repair performance. In practice, each stripe is stored on a different set of storage nodes spanning across the network. Our goal is to distribute the load of a multi-stripe repair across all available helpers as evenly as possible.

We adopt a simple greedy scheduling approach for the selection of helpers. For each node in the storage system, repair pipelining keeps track of a timestamp indicating when the node was last selected as a helper for a single-stripe repair. To repair a failed block of a stripe, we select k out of $n-1$ available helpers in the stripe that have the smallest timestamps; in other words, the k se-

lected helpers are the least recently selected in previous requests. Choosing the k out of the $n-1$ helpers can be done in $O(n)$ time using the quick select algorithm [13] (based on repeated partitioning of quick sort). We use a centralized coordinator to manage the selection process (§5). Our greedy scheduling emphasizes simplicity in deployment. We can also adopt a more sophisticated approach by weighting node preferences in real time [20].

Unlike the degraded read scenario, the multiple reconstructed blocks can be stored on multiple requestors. Under this condition, the gain of repair pipelining over conventional repair decreases, as the latter can also parallelize the repair across multiple requestors. Nevertheless, our evaluation indicates that repair pipelining still provides repair performance improvements (§6).

Note that the number of requestors that can be selected and the choices of requestors may also depend on various deployment factors [20]. In this work, we assume that the requestors are selected offline in advance.

4 Heterogeneity

In practice, the links of a distributed storage system have different bandwidths [9, 18]. We now extend the design of repair pipelining in §3 in two aspects: (i) a requestor can read slices from multiple helpers in parallel, and (ii) we solve a weighted path selection problem to find an optimal path of k helpers that maximizes repair performance. Each extension addresses a different heterogeneous setting.

4.1 Parallel Reads

In the original design of repair pipelining, a requestor always reads slices from one helper. This may lead to last-mile congestion. For example, a client (requestor) sits at the network edge and accesses a cloud storage system that is far from the client. We propose a *cyclic version* of repair pipelining that allows a requestor to read slices from multiple helpers.

We now describe the cyclic version. Our discussion assumes that all links are homogeneous and it takes one timeslot to transmit a block size of data in a link. The cyclic version again divides a failed block into s fixed-size slices S_1, S_2, $\cdots$, S_s, and repairs each slice through some linear path to eliminate any bottlenecked link. However, it now maps the k helpers N_1, N_2, $\cdots$, N_k into different *cyclic paths* that can be cycled from N_k through N_1. Specifically, it partitions the s slices into $\lceil \frac{s}{k-1} \rceil$ groups, each of which has $k-1$ slices (the last group has fewer than $k-1$ slices if s is not divisible by $k-1$). The repair of each group of slices is then performed in two phases. Without loss of generality, we only consider how to repair the first group S_1, S_2, $\cdots$, S_{k-1}. In the first phase, repairing each slice S_i $(1 \le i \le k-1)$ traverses through the cyclic path

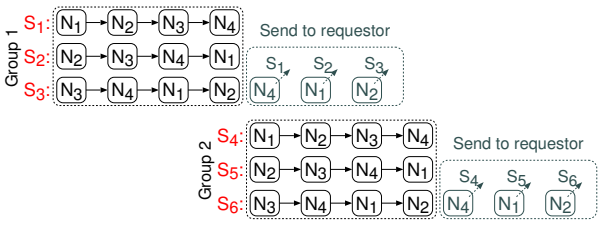


Figure 4: Cyclic version of repair pipelining with $k=4$ and $s=6$.

$N_i \to N_{i+1} \to \cdots N_k \to N_1 \to \cdots N_{i-1}$. We repair all slices through different cyclic paths simultaneously, and each slice-level transmission takes $\frac{1}{s}$ timeslots. The first phase can be done in $\frac{k-1}{s}$ timeslots. In the second phase, the last helper of each cyclic path delivers the repaired slice to the requestor. The second phase is also done in $\frac{k-1}{s}$ timeslots. Figure 4 shows the cyclic version for $k=4$ and $s=6$.

Note that we can start repairing the slices of the next group simultaneously while we deliver the repaired slices for the current group. Specifically, while $k-1$ helpers simultaneously transmit slices for the repair in the next group, there is one idle helper that can transmit the repaired slice for the current group to the requestor. They can be done together in $\frac{k-1}{s}$ timeslots.

We analyze the time complexity of the cyclic version under the homogeneous link assumption. We only consider the case where s is divisible by $k-1$, while the same result can be derived otherwise. Repairing each group of slices takes $\frac{2(k-1)}{s}$ timeslots, and the repair of the last group starts after $(\frac{s}{k-1}-1)\frac{k-1}{s}$ timeslots. The whole repair time is $(\frac{s}{k-1}-1)\frac{k-1}{s} + \frac{2(k-1)}{s} = 1 + \frac{k-1}{s} \to 1$, as s is sufficiently large.

Note that the cyclic version now allows a requestor to read slices from $k-1$ helpers. If the repair bottleneck lies on the network transfer from the helpers to the requestor, our evaluation shows that the cyclic version significantly outperforms the original design of repair pipelining (§6).

4.2 Weighted Path Selection

We now study a more general heterogeneous setting in which link bandwidths can have arbitrary values. To motivate, we consider geo-distributed data centers that span multiple geographic regions [1, 10]. They typically stripe redundancy across regions to protect against large-scale correlated failures. However, intra- and inter-region bandwidths are highly different. Table 1 shows one of our `iperf` [16] measurement tests for the intra- and inter-region bandwidths on Amazon EC2 across four regions respectively in North America and Asia. We observe that intra-region bandwidths are in general more abundant than inter-region bandwidths, and inter-region bandwidths have a high degree of variance.

Table 1: A test of intra- and inter-region bandwidth measurements (in Mb/s) on Amazon EC2 in North America and Asia. Each number is the measured bandwidth from the row region to the column region.

(a) North America

Bandwidth	California	Canada	Ohio	Oregon
California	501.3	57.2	44.1	299.9
Canada	55.3	732.0	63.3	48.0
Ohio	46.3	65.7	332.5	95.6
Oregon	297.8	50.2	93.6	250.1

(b) Asia

Bandwidth	Mumbai	Seoul	Singapore	Tokyo
Mumbai	624.8	62.3	39.5	37.7
Seoul	63.8	265.7	86.1	183.2
Singapore	41.5	88.1	493.0	49.1
Tokyo	39.7	181.0	46.9	489.1

In the following, we extend repair pipelining to solve a *weighted path selection* problem. We focus on extending the design for the single-block repair (of a single stripe) for degraded reads (§3.2). We later discuss how our extended design is applied to full-node recovery (§3.3).

4.2.1 Formulation

Recall that for a single-block repair, repair pipelining transmits a number of slices along a path of k helpers, say $N_1 \rightarrow N_2 \rightarrow \cdots \rightarrow N_k \rightarrow R$. Suppose that the link bandwidths are different. If the number of slices is sufficiently large, then the slices are transmitted in parallel through the path (Figure 3), and the performance of repair pipelining will be bottlenecked by the link with the minimum available bandwidth along the path. To minimize the single-block repair time, we should find a path that maximizes the minimum link bandwidth. Here,

To repair a failed block, we need to find k out of $n-1$ available helpers of the same stripe as the failed block, and also find the sequence of link transmissions so that the path along the k selected helpers and the requestor minimizes the single-block repair time. Specifically, there are a total of n nodes, including the $n-1$ available helpers and the requestor. We associate a *weight* with each (directed) link from one node to another node, such that a higher weight implies a longer transmission time along the link. For example, the weight can be represented by the inverse of the link bandwidth obtained by periodic measurements on link utilizations [5]. Then our objective is to find a path of $k+1$ nodes (i.e., k selected helpers and the requestor) that minimizes the maximum link weight of the path. Here, we focus on link weights, and the same idea is applicable if we associate weights with nodes. Any straggler is assumed to be associated with a large weight, so it will be excluded from the selected path.

To solve the above problem, a naïve approach is to perform a brute-force search on all possible candidate paths. However, there are a total of $\frac{(n-1)!}{(n-1-k)!}$ permutations, and the brute-force search becomes computationally expensive even for moderate sizes of n and k. Since the link weights vary over time, the path selection should be done quickly on-the-fly based on the measured link weights.

Algorithm 1 Weighted Path Selection

```
Input: link weights
Output: optimal path P*
 1: procedure MAIN
 2:   P = R
 3:   P* = null
 4:   w* = ∞
 5:   N = set of n − 1 available helpers
 6:   EXTENDPATH
 7:   return P*
 8: end procedure
 9: function EXTENDPATH
10:   if P.length < k + 1 then
11:     for each node N ∈ N not in P do
12:       if weight(N, head node of path P) < w* then
13:         P = N → P
14:         EXTENDPATH
15:         remove N from P
16:       end if
17:     end for
18:   else
19:     P* = P
20:     w* = maximum link weight of P
21:   end if
22: end function
```

4.2.2 Algorithm

We present a fast yet optimal algorithm that quickly identifies an optimal path. The algorithm builds on brute-force search to ensure that all candidate paths are covered, but eliminates the search of infeasible paths. Our insight is that if a link L has a weight larger than the maximum weight of an optimal path candidate that is currently found, then we no longer need to search for the paths containing link L, since the maximum weight of any path containing L must be larger than the maximum weight of the optimal path candidate.

Algorithm 1 shows the pseudo-code of the weighted path selection algorithm. Let P be the path that we currently consider, P^* be the optimal path candidate that we have found, w^* be the maximum link weight of P^*, and $\mathcal{N}$ be the set of $n-1$ available helpers. We first initialize a path P with only the requestor R (Line 2), such that R will be the tail node of P. We also initialize P^*, w^*, and $\mathcal{N}$ (Lines 3-5). We call the recursive function EXTENDPATH (Line 6) and finally return the optimal path P^* (Line 7).

The function EXTENDPATH recursively extends P by

one node in $\mathcal{N}$ and appends the node to the head of P if the link weight from the node to the current head node of P is less than w^*; otherwise, the path containing the link cannot minimize the maximum link weight as argued above. Specifically, the algorithm appends $N \in \mathcal{N}$ to P if the current path length is less than $k+1$ and the weight from N to the head node of P is less than w^* (Lines 10-13). It calls EXTENDPATH again to consider candidate paths that now include $N \rightarrow P$ (Line 14). It then removes N from P (Line 15), and tries other nodes in $\mathcal{N}$. If the length of P is now $k+1$, it implies that all of its links have weight less than w^*, so we update P as the new optimal path P^* and w^* as the maximum link weight of P^* (Lines 19-20).

Algorithm 1 significantly reduces the search time. We evaluate the search time for (14,10) codes using Monte-Carlo simulations over 1,000 runs on a machine with 3.7 GHz Intel Xeon E5-1620 v2 CPU and 16 GiB memory. The brute-force search takes 27s on average, while Algorithm 1 reduces the search time to only 0.9ms.

4.2.3 Discussion

Algorithm 1 also addresses full-node recovery (§3.3). Specifically, we apply Algorithm 1 to each stripe. If we apply greedy scheduling on helper selection, we simply substitute $\mathcal{N}$ with the set of k selected helpers. Note that the brute-force search for the optimal path on the k selected helpers remains expensive, since it still needs to consider $k!$ permutations on the sequence of link transmissions along the path. Thus, Algorithm 1 still significantly saves the search time in this case.

We can also apply Algorithm 1 to the cyclic version in §4.1. Instead of searching for an optimal path, we now search for an optimal cycle of k helpers that minimizes the maximum link weight.

5 Implementation

We implemented a prototype called ECPipe to realize repair pipelining. ECPipe runs as a middleware atop an existing storage system and performs repair operations on behalf of the storage system. Moving the repair logic to ECPipe greatly reduces changes to the code base of the storage system to realize new repair techniques, while we focus on optimizing ECPipe to maximize the repair performance gain. We have integrated ECPipe with two widely deployed distributed storage systems HDFS [36] and QFS [24]. HDFS is written in Java, while QFS is written in C++. Our ECPipe prototype is mostly written in C++, and the part for HDFS integration is in Java. Our ECPipe prototype has around 3,000 lines of code.

5.1 Erasure Coding in HDFS and QFS

HDFS: Erasure coding in HDFS is done by the HDFS-RAID module [12]. HDFS-RAID deploys a *RaidNode* atop HDFS for erasure coding management. HDFS initially stores data as fixed-size blocks (64 MiB by default) with replication; later, the RaidNode encodes replicated blocks into coded blocks via MapReduce [7]. The RaidNode also checks for any lost or corrupted coded block (by verifying block checksums). If so, it repairs the failed blocks, either by itself in local mode or via a MapReduce job in distributed mode. Both modes will issue reads to k available blocks of the same stripe in parallel from HDFS, reconstruct the failed block, and write back to HDFS. HDFS-RAID also provides a RAID file system client to access coded blocks. For a degraded read to a failed block, the RAID file system reads k available blocks of the same stripe in parallel and reconstructs the failed block.

QFS: Different from HDFS, which stores data with both replication and erasure coding, QFS stores all data in erasure-coded format. QFS supports (9,6) RS codes [32]. The QFS client writes data into six 1 MiB buffers. When the buffers fill up, it encodes the six 1 MiB buffers into three 1 MiB parity buffers. It then appends the nine 1MB buffers to nine data and parity blocks (the default block size is 64 MiB) that are stored in nine storage nodes. To repair any failed block, a storage node retrieves six available blocks from other storage nodes for reconstruction.

5.2 ECPipe Design

Figure 5 shows the ECPipe architecture. It uses a *coordinator* to manage the repair operation between a requestor and multiple helpers. ECPipe runs on top of a storage system. To repair a failed block, the storage system creates a requestor object, which sends a repair request with the failed block ID to the coordinator (step 1). The coordinator uses the failed block ID to identify the locations of k available blocks of the same stripe. It notifies all helpers with the block locations (step 2). The helpers retrieve the blocks, perform repair pipelining in slices, and deliver the repaired slices to the requestor (step 3).

We integrate ECPipe with a storage system in three aspects. First, we implement the requestor as a class (in C++ and Java) that can be instantiated by the storage system to reconstruct failed blocks. For HDFS, the requestor is created in either the RaidNode or the RAID file system client; for QFS, it is created by the storage node that starts a repair operation. Second, we implement each helper as a daemon that is co-located with each storage node to directly read the locally stored blocks. Our insight is that both HDFS and QFS store each block in the underlying native file system as a plain file, and use the block ID to form the file name. Thus, each helper can directly read the stored blocks through the native file system. This eliminates the need of helpers to fetch data through the distributed storage system routine. It not only reduces the burden of metadata management of the

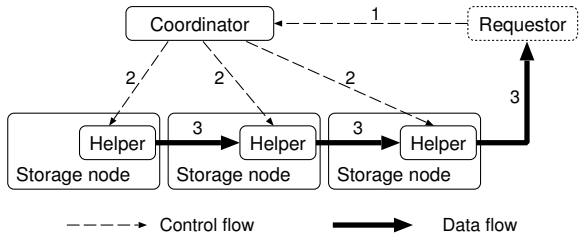


Figure 5: ECPipe architecture.

distributed storage system, but also improves repair performance (§6.3). Finally, the coordinator needs to access both block locations and the mappings of each block to its stripe. For HDFS, we retrieve the information from the RaidNode; for QFS, we retrieve the information from a storage node when it starts a repair operation.

To simplify our implementation, ECPipe uses Redis [31] to pipeline slices across helpers. Each helper maintains an in-memory key-value store based on Redis, and uses the client interface of Redis to transmit slices among helpers. In addition, each helper performs disk I/O, network transfer, and computation via multiple threads for performance speedup. Adding ECPipe into HDFS and QFS only requires changes of around 110 and 180 lines of code, respectively.

To provide fair comparisons (§6), we also implement conventional repair (§2.2) and PPR [20] under the same ECPipe framework, by only changing the transmission flow of data during repair.

6 Evaluation

We conducted experiments on a local cluster and two geo-distributed clusters deployed on Amazon EC2. We show that repair pipelining outperforms both conventional repair and PPR [20], for both degraded reads and full-node recovery.

6.1 ECPipe Performance on a Local Cluster

6.1.1 Methodology

We first evaluate ECPipe when it runs as a standalone system. We conducted experiments on a local cluster of 19 machines, each of which has a quad-core 3.1 GHz Intel Core i5-2400 CPU, 8 GiB RAM, and a Seagate ST31000524AS 1 TiB SATA hard disk. We host the coordinator on one machine, and 18 helpers on the remaining machines. All machines are connected via a 1 Gb/s Ethernet switch. The 1 Gb/s bandwidth can be viewed as modeling the cross-rack bandwidth available for repair tasks in a production cluster [34], in which the blocks of a stripe are stored in distinct racks and there will be cross-rack transfers during repair.

Initially, we store coded blocks in the local file system of each machine, and load block locations and stripe information into the coordinator. We simulate a "failed" machine by erasing blocks there, and repair the failed block of each stripe on a requestor. To fairly evaluate the impact of network transfers on repair, we host the requestor on a machine that does not store any available block of the repaired stripe, so as to ensure that the available blocks are always transmitted over the network. By default, we configure 64 MiB block size, 32 KiB slice size (for repair pipelining only), and (14,10) RS codes; note that (14,10) RS codes are also used by Facebook [30, 34]. We vary one of the settings at a time and evaluate its impact.

We consider two versions of repair pipelining: the basic version in §3 and the cyclic version in §4.1. We compare them with conventional repair (§2) and PPR [20].

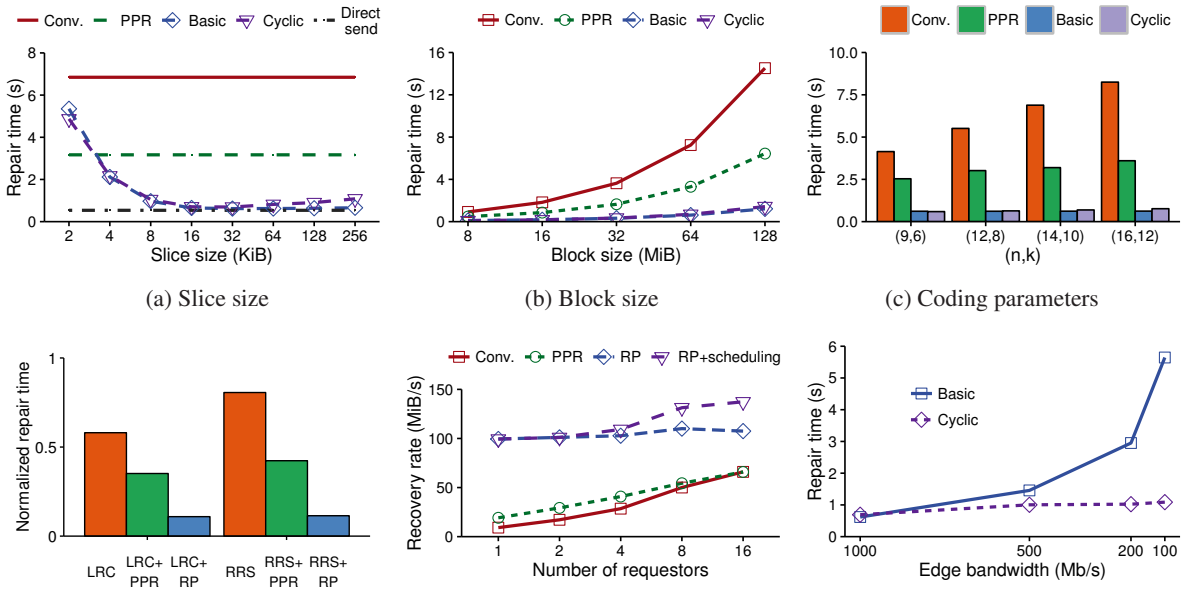
We evaluate both degraded reads and full-node recovery. For degraded reads (Figures 6(a)-(d) and 6(f)), we measure the *single-block repair time*, defined as the latency from issuing a degraded read request to a failed block until the block is reconstructed. For full-node recovery (Figure 6(e)), we measure the *recovery rate*, defined as the amount of recovered data by the total repair time. All results are averaged over 10 runs. The standard deviations are small and hence omitted from the plots.

6.1.2 Results

Slice size: Figure 6(a) shows the single-block repair time versus the slice size in repair pipelining. It also plots the transmission time of directly sending a single block over a 1 Gb/s link (labeled as "Direct send"). Both basic and cyclic versions of repair pipelining have high repair times when the slice size is small, even though more slices are pipelined during a repair (i.e., s is large). The reason is that the overhead of issuing transmission requests for many slices becomes significant. Nevertheless, the repair times of both versions decrease as the slice size increases up to 32 KiB (where $s = 2{,}048$) since fewer transmission requests are issued, and then increase since there are fewer slices in a block being pipelined. When the slice size is 32 KiB, the basic version reduces the single-block repair time by 90.9% and 80.4% compared to conventional repair and PPR, respectively. The basic version achieves 10.7% less single-block repair time than the cyclic version. The reason is that a helper in the cyclic version sends data to both the requestor as well as its next-hop helper, so the two transfers interfere with one another and (slightly) increase the repair time.

Also, the direct send time of transferring a 64 MiB block is 0.57s, which is almost network-bound in our 1 Gb/s network. The single-block repair time of the basic version is only 7.0% more than the direct send time, showing the feasibility of achieving $O(1)$ repair time.

Block size: Figure 6(b) shows the single-block repair time versus the block size. The repair time reduction of



(a) Slice size (b) Block size (c) Coding parameters

(d) Repair-friendly codes (e) Full-node recovery (f) Limited edge bandwidth

Figure 6: ECPipe performance on a local cluster.

repair pipelining over conventional repair and PPR increases with the block size as it can partition a block into more slices for better network usage. The basic version of repair pipelining reduces the single-block repair time by up to 91.4% and 80.9% compared to conventional repair and PPR, respectively. It is also faster than the cyclic version by up to 16.7%.

Coding parameters: Figure 6(c) shows the single-block repair time versus (n, k). The single-block repair times of both conventional repair and PPR increase with k, while that of repair pipelining is almost unchanged. As k increases from 6 to 12, the repair time reduction of the basic version increases from 85.1% to 92.1% compared to conventional repair, and from 75.7% to 83.3% compared to PPR.

Repair-friendly codes: We demonstrate how repair pipelining is compatible with practical erasure codes. We consider two state-of-the-art repair-friendly codes: LRC [15] and Rotated RS codes [17]. LRC has higher storage redundancy than RS codes by associating local parity blocks with a subset of data blocks, so as to improve single-block repair performance. On the other hand, rotated RS codes arrange the layout of parity blocks to improve the performance of a degraded read to a series of data blocks. We configure LRC with $k = 12$ data blocks, and Rotated RS codes with $(n, k) = (16,12)$. LRC needs to read only six blocks (five data blocks plus one local parity block) for repairing a failed data block, while Rotated RS codes on average read nine blocks for repairing a failed data block. Here, we focus on the basic version of repair pipelining.

Figure 6(d) shows the normalized single-block repair time with respect to the conventional repair of (16,12) RS codes. Although repair pipelining does not reduce the amount of repair traffic as in LRC and Rotated RS codes, its normalized repair time (around 0.1) is much smaller than those of LRC and Rotated RS codes by effectively utilizing the bandwidth resources of all helpers. We observe the same improvement in PPR, but its repair time reduction is less than that of repair pipelining.

Full-node recovery: We now evaluate full-node recovery with multiple requestors and our greedy scheduling in helper selection (§3.3). We randomly write multiple stripes of blocks across all 18 helpers in the local cluster. We erase 64 blocks from 64 stripes (one block per stripe) in one helper to mimic a single node failure, and recover all the erased blocks simultaneously. We distribute the reconstructed blocks evenly across a number of requestors (i.e., 1, 2, 4, 8, and 16).

We consider two cases of helper selection based on the basic version of repair pipelining: (i) we index the helpers from 1 to 18, and always select the available blocks from the k helpers that have the smallest indexes in a stripe for repair (labeled as "RP"); and (ii) we use the greedy approach to select k helpers that are least recently accessed for repair (labeled as "RP+scheduling"). We also evaluate conventional repair and PPR, both of which select helpers as in RP without greedy scheduling.

Figure 6(e) shows the recovery rates. As the number of requestors increases, the recovery rates of all schemes increase. Conventional repair sees the largest gain by distributing the repair load across more requestors. Its

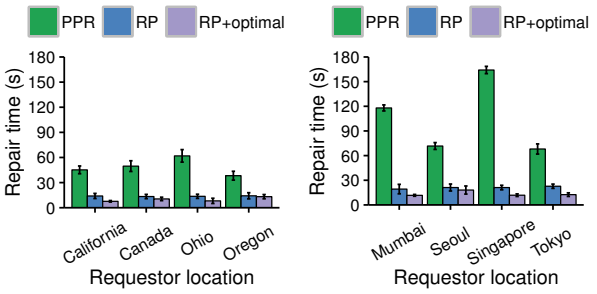
performance is also close to that of PPR as the number of requestors increases. However, repair pipelining still outperforms conventional repair by making bandwidth utilization more balanced. Furthermore, our greedy scheduling achieves a higher gain when there are more requestors by better distributing the repair load across all helpers. For example, when there are 16 requestors, the recovery rate of repair pipelining without greedy scheduling is 1.63× that of conventional repair, and our greedy scheduling further improves the recovery rate of repair pipelining by 27.9%.

Limited edge bandwidth: In previous tests, the basic version of repair pipelining always outperforms the cyclic version. We now show the benefits of the cyclic version when a requestor sits at the network edge and the edge bandwidth from the storage system to the requestor is limited (§4.1). We use the Linux command `tc` [38] to limit the edge bandwidth from each helper to the requestor. Figure 6(f) shows the single-block repair time versus the edge bandwidth. As the edge bandwidth decreases, the repair time of the basic version increases significantly, while that of the cyclic version only increases mildly by allowing the requestors to read repaired data from multiple helpers in parallel. For example, the cyclic version has 80.1% less repair time than the basic version when the edge bandwidth is 100 Mb/s.

6.2 ECPipe Performance on Amazon EC2

Methodology: We evaluate ECPipe on two independent Amazon EC2 clusters, one in North America and one in Asia. Each cluster is deployed in four regions as shown in Table 1. We deploy four EC2 instances per region per cluster to host helpers (i.e., 16 helpers in total), and one EC2 instance in Ohio and Singapore to host the coordinator for the North America and Asia clusters, respectively. Note that the overhead of accessing the coordinator has negligible impact on the overall repair performance. We focus on evaluating the degraded reads (in terms of single-block repair time) issued by a requestor. We host the requestor on an EC2 instance in each region and study how the performance varies across regions. All EC2 instances are of type `t2.micro`.

We configure 64 MiB block size and 32 KiB slice size for repair pipelining. We use (16,12) RS codes and distribute the 16 blocks of each stripe across the 16 EC2 instances in four regions; this also provides fault tolerance against any single-region failure. We consider two versions of repair pipelining: the basic version in §3 (labeled as "RP"), which finds a random path across k randomly selected helpers, and the optimal version in §4.2 (labeled as "RP+optimal"), which finds an optimal path via Algorithm 1. Note that the network bandwidth fluctuates over time, although intra-region bandwidth remains higher than inter-region bandwidth, as shown in Table 1.



(a) North America cluster (b) Asia cluster

Figure 7: ECPipe performance on Amazon EC2.

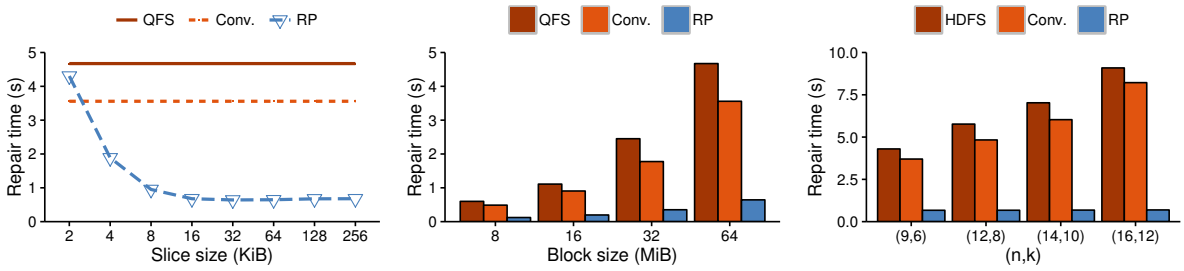
Thus, the optimal version probes the network bandwidth via `iperf` before each run of experiments. We average our results over 10 runs, and also include the standard deviations as the results have higher variances than in our local cluster.

Results: Figure 7 shows the single-block repair times and the standard deviations of PPR and the two versions of repair pipelining in both clusters; we do not show the results of conventional repair, whose repair time goes beyond 200s. Repair pipelining (without weighted path selection) achieves repair time saving over PPR in all cases when the requestor is in different regions. The repair time reduction is 62.7-78.0% for North America and 66.6-87.1% for Asia. Our weighted path selection further reduces the repair time by 7.3-45.4% for North America and 14.5-45.0% for Asia, compared to repair pipelining without weighted path selection. Note that our weighted path selection can be done in around 1ms (§4.2), which is negligible compared to the repair time in our evaluation.

6.3 Performance on HDFS and QFS

Methodology: We evaluate the integration of ECPipe into HDFS and QFS, both of which are deployed on our local cluster (§6.1). We co-locate a helper daemon with each storage node (18 nodes in total). By default, we set the slice size of repair pipelining as 32 KiB and block size as 64 MiB. For QFS, we use its default (9,6) RS codes and vary the slice size and block size. For HDFS, we vary (n, k). We consider three repair schemes: (i) the original repair implementations of HDFS and QFS, both of which are based on conventional repair, (ii) the conventional repair under ECPipe, and (iii) the basic version of repair pipelining in §3 under ECPipe. We evaluate degraded reads (in terms of single-block repair time) issued by a requestor that is attached with either an HDFS or QFS client. We report averaged results over 10 runs as in §6.1 (the standard deviations are small and omitted).

Results: Figure 8 shows the evaluation results. First, repair pipelining under ECPipe significantly improves the repair performance of the original repair implementa-



(a) QFS: slice size (b) QFS: block size (c) HDFS: coding parameters

Figure 8: Performance on HDFS and QFS.

tions of HDFS and QFS. It reduces the single-block repair time by up to 86.3% when the slice size is 32 KiB and the block size is 64 MiB (Figures 8(a) and 8(b)), and by 84.4–92.4% for different coding parameters (Figure 8(c)). The results are consistent with those in §6.1.

We observe that moving the repair logic to ECPipe improves repair performance. Specifically, conventional repair under ECPipe reduces the single-block repair time by up to 16.2% and 23.8% in HDFS and QFS, respectively, compared to the original conventional repair implementation. The reason of the performance gain is that the helpers of ECPipe can directly access the stored blocks via the native file system, instead of fetching the blocks through the distributed storage system routine. Nevertheless, we emphasize that the repair performance gain mainly comes from repair pipelining, rather than the implementation of ECPipe. Although moving repair to ECPipe reduces repair time, the reduction is minor compared to the reduction achieved by repair pipelining.

7 Related Work

Many new erasure codes have been proposed to mitigate repair overhead, especially for single-node repair. To name a few, regenerating codes [8] minimize repair traffic by allowing storage nodes to send encoded data for repair. Rotated RS codes [17] reduce repair traffic and disk I/O of a degraded read to a sequence of data blocks. Hitchhiker [30] extends RS codes [32] to piggyback parity information of one stripe into another stripe, and is shown to reduce both bandwidth and I/O for repair by up to 45%. PM-RBT codes [28] are special regenerating codes that simultaneously minimize bandwidth, I/O, and storage redundancy. Butterfly codes [25] are systematic regenerating codes that provide double-fault tolerance. Locally repairable codes [15, 34] add local parity blocks to mitigate repair I/O with extra storage.

Instead of constructing new erasure codes, we design new repair strategies for general practical erasure codes (including repair-friendly codes). Some prior studies are also along this direction. Lazy repair [3, 37] defers immediate repair action until a tolerable limit is reached. To speed up full-node recovery, the repair of multiple stripes can be parallelized across available nodes, as also adopted by replicated storage [6, 23] and de-clustered RAID arrays [14]. Degraded-first scheduling [19] targets MapReduce on erasure-coded storage by scheduling map tasks to fully utilize bandwidth in degraded reads. CAR [35] focuses on RS codes in data centers, and computes partial repaired results in each rack to mitigate cross-rack repair traffic. The most closely related work to ours is PPR [20], which reduces repair time from $O(k)$ to $O(\log k)$. Repair pipelining further reduces it to $O(1)$. We also show how repair pipelining addresses heterogeneous environments with different link bandwidths.

8 Conclusions

Repair pipelining is a general technique to reduce the repair time to almost the same as the normal read time in erasure-coded storage. It pipelines the repair of a failed block across storage nodes in units of slices, so as to evenly distribute repair traffic and fully utilize bandwidth resources across storage nodes. Our contributions include: (i) the design of repair pipelining for both degraded reads and full-node recovery, (ii) the extensions of repair pipelining with parallel reads and weighted path selection for heterogeneous environments, (iii) a repair prototype ECPipe and its integrations into HDFS and QFS, and (iv) experiments that show the repair speedup through repair pipelining on a local cluster and Amazon EC2. The source code of our ECPipe prototype is available at: **http://adslab.cse.cuhk.edu.hk/software/ecpipe**.

Acknowledgments: We thank our shepherd, Ryan Huang, and the anonymous reviewers for their valuable comments. We thank Allen Poon for contributing to the early implementation. This work was supported in part by the Research Grants Council of Hong Kong (GRF 14216316 and CRF C4047-14E), VC Discretionary Fund of CUHK (VCF2014007), and Cisco University Research Program Fund (CG#593756) from Silicon Valley Community Foundation.

References

[1] M. K. Aguilera. Geo-distributed Storage in Data Centers. In *Slides presented at OPODIS*, 2013.

[2] F. André, A.-M. Kermarrec, E. L. Merrer, N. L. Souarnec, G. Straub, and A. van Kempen. Archiving Cold Data in Warehouses with Clustered Network Coding. In *Proc. of ACM EuroSys*, Apr 2014.

[3] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. Voelker. Total Recall: System Support for Automated Availability Management. In *Proc. of NSDI*, 2004.

[4] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proc. of ACM SOSP*, Oct 2011.

[5] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging Endpoint Flexibility in Data-Intensive Clusters. In *Proc. of ACM SIGCOMM*, Aug 2013.

[6] B. 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 *Proc. of NSDI*, 2006.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of USENIX OSDI*, Dec 2004.

[8] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. Network Coding for Distributed Storage Systems. *IEEE Trans. on Info. Theory*, 56(9):4539–4551, Sep 2010.

[9] T. Ernvall, S. El Rouayheb, C. Hollanti, and H. V. Poor. Capacity and Security of Heterogeneous Distributed Storage Systems. *IEEE Journal on Selected Areas in Communications*, 31(12):2701–2709, Dec 2013.

[10] D. Ford, F. Labelle, F. I. Popovici, M. Stokel, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proc. of USENIX OSDI*, Oct 2010.

[11] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In *Proc. of ACM SOSP*, Dec 2003.

[12] HDFS-RAID. `http://wiki.apache.org/hadoop/HDFS-RAID`.

[13] C. A. R. Hoare. Algorithm 65: find. *Communications of the ACM*, 4(7):321–322, 1961.

[14] M. Holland and G. A. Gibson. Parity Declustering for Continuous Operation in Redundant Disk Arrays. In *Proc. of ASPLOS*, 1992.

[15] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure Coding in Windows Azure Storage. In *Proc. of USENIX ATC*, Jun 2012.

[16] Iperf. `https://iperf.fr/`.

[17] 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 *Proc. of USENIX FAST*, Feb 2012.

[18] J. Li, S. Yang, X. Wang, and B. Li. Tree-structured Data Regeneration in Distributed and Storage Systems with Regenerating Codes. In *Proc. of IEEE INFOCOM*, 2010.

[19] R. Li, P. P. C. Lee, and Y. Hu. Degraded-First Scheduling for MapReduce in Erasure-Coded Storage Clusters. In *Proc. of IEEE/IFIP DSN*, 2014.

[20] S. Mitra, R. Panta, M.-R. Ra, and S. Bagchi. Partial-Parallel-Repair (PPR): A Distributed Technique for Repairing Erasure Coded Storage. In *Proc. of ACM EuroSys*, 2016.

[21] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. f4: Facebook's Warm BLOB Storage System. In *Proc. of USENIX OSDI*, 2014.

[22] L. M. Ni and P. K. McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *IEEE Computer*, 26(2):62–76, Feb 1993.

[23] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proc. of ACM SOSP*, 2011.

[24] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The Quantcast File System. In *Proc. of VLDB Endowment*, 2013.

[25] L. Pamies-Juarez, F. Blagojević, R. Mateescu, C. Gyuot, E. E. Gad, and Z. Bandic. Opening the Chrysalis: On the Real Repair Performance of MSR Codes. In *Proc. of USENIX FAST*, Feb 2016.

[26] J. S. Plank. Erasure Codes for Storage Systems: A Brief Primer. *;login: the Usenix magazine*, 38(6):44–50, Dec 2013.

[27] 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 *Proc. of USENIX FAST*, 2009.

[28] 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 *Proc. of USENIX FAST*, 2015.

[29] 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 *Proc. of USENIX HotStorage*, 2013.

[30] K. V. 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 *Proc. of ACM SIGCOMM*, 2014.

[31] Redis. `http://redis.io/`.

[32] I. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[33] J. K. Resch and J. S. Plank. AONT-RS: Blending Security and Performance in Dispersed Storage Systems. In *Proc. of USENIX FAST*, 2011.

[34] 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 *Proc. of VLDB Endowment*, pages 325–336, 2013.

[35] Z. Shen, J. Shu, and P. P. C. Lee. Reconsidering Single Failure Recovery in Clustered File Systems. In *Proc. of IEEE/IFIP DSN*, 2016.

[36] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proc. of IEEE MSST*, May 2010.

[37] M. Silberstein, L. Ganesh, Y. Wang, L. Alvizi, and M. Dahlin. Lazy Means Smart: Reducing Repair Bandwidth Costs in Erasure-coded Distributed Storage. In *Proc. of ACM SYSTOR*, 2014.

[38] `tc`. `https://linux.die.net/man/8/tc`.

[39] H. Weatherspoon and J. D. Kubiatowicz. Erasure Coding vs. Replication: A Quantitative Comparison. In *Proc. of IPTPS*, Mar 2002.

PARIX: Speculative Partial Writes in Erasure-Coded Systems

Huiba Li
mos.meituan.com

Yiming Zhang
NUDT

Zhiming Zhang
mos.meituan.com

Shengyun Liu
NUDT

Dongsheng Li
NUDT

Xiaohui Liu
NUDT

Yuxing Peng
NUDT

Abstract

Erasure coding (EC) has been widely used in cloud storage systems because it effectively reduces storage redundancy while providing the same level of durability. However, EC introduces significant overhead to small write operations which perform *partial write* to an entire EC group. This has been a major barrier for EC to be widely adopted in small-write-intensive systems such as virtual disk service. Parity logging (PL) appends parity changes to a journal to accelerate partial writes. However, since previous PL schemes have to perform a time-consuming write-after-read for each partial write, i.e., read the current value of the data and then compute and write the *parity delta*, their write performance is still much lower than that of replication-based storage.

This paper presents PARIX, a speculative partial write scheme for fast parity logging. We transform the original formula of parity calculation, so as to use the *data* deltas (between the current/original data values), instead of the *parity* deltas, to calculate the parities during journal replay. For each partial write, this allows PARIX to *speculatively* log only the current value of the data. The original value is needed only once in a journal when performing the first write to the data. For a series of n partial writes to the same data, PARIX performs pure write (instead of write-after-read) for the last $n-1$ ones while only introducing a small penalty of an extra network RTT (round-trip time) to the first one. Evaluation results show that PARIX remarkably outperforms state-of-the-art PL schemes in partial write performance.

1 Introduction

Failures are common in large-scale cloud storage systems [22, 34, 35]. For example, more than 1000 server failures occur in one year in Google's 1800-server clusters [5]. To maintain data durability against failures, storage systems usually have two options, namely, replication [24] and erasure coding (EC) [25]. In replication, the storage system uses multiple replicas for each piece of data, while EC encodes the original data to generate new parities such that the original data can be recovered from a subset of the data and parities. EC has less storage overhead than replication while providing the same or even higher level of durability [32], and thus has been widely adopted in not only RAID systems [10, 30, 28, 20] but also modern cloud storage systems [13, 16, 27].

In cloud storage systems like Amazon Dynamo [12] and Windows Azure [7], small write operations [29] (which perform *partial write* to an entire EC group) are dominant for many real-world workloads. For erasure-coded storage systems that frequently perform small writes, it is important to efficiently support EC partial writes. Usually there are two ways to perform writes [8], namely, in-place update which directly updates the new data, and log-based update which appends the writes to a journal [26]. The logs are asynchronously *replayed* to update the data with the latest values when the system is idle.

Logging improves the write performance but degrades the read performance [32]. Parity logging (PL) [29] adopts a hybrid approach. Since normally only the data is read and the parities will only be read when the data is not available, PL respectively performs in-place update and log-based update for writes of the data and of the parities, so as to achieve a balance between reads and writes. However, state-of-the-art PL schemes [29, 17, 8] have to perform a time-consuming write-after-read for each partial write to compute the *parity delta* (which will be used to "patch" the parity during journal replay), and thus their write performance is still significantly lower than that of replication [32].

This paper presents PARIX, a speculative partial write scheme for fast parity logging. We transform the original formula of parity calculation, so as to use the *data* deltas (between the current and original values), instead of the *parity* deltas, to update the parities during journal replay. For each partial write, this allows PARIX to *speculatively*

log only the new value of the data without reading its original value, which is needed only once in a journal when performing the first write to the data. For a series of n partial writes to the same data, PARIX performs pure write (instead of write-after-read) for the last $n-1$ ones while only introducing a small penalty of an extra network RTT (round-trip time) to the first one.

Based on PARIX, we have built a prototype of an erasure-coded block store [1] providing virtual disks that can be mounted by cloud-oblivious applications with strong consistency guarantees. Evaluation on the PARIX block store shows that PARIX not only achieves similar or even higher I/O performance compared to replication (with much higher storage efficiency), but also remarkably outperforms state-of-the-art PL schemes in partial write performance by up to orders of magnitude.

This paper makes the following contributions.

- We propose a novel speculative partial write scheme (PARIX) for fast parity logging in erasure-coded storage systems.
- We apply PARIX and implement an erasure-coded block store supporting efficient journal replay and fast failure recovery.
- We report evaluation results of PARIX's I/O performance from prototype measurement to demonstrate the effectiveness of our designs.

The rest of this paper is organized as follows. §2 discusses the background and related work. §3 introduces PARIX partial writes. §4 describes the prototype of a block store using PARIX-backed EC. §5 presents the evaluation results. And §6 concludes the paper.

2 Background

Erasure coding (EC) introduces less storage overhead than replication while providing the same level of durability [8]. Essentially, EC calculates linear combinations of the original data in the Galois Field [25] $GF(2^{\wedge}w)$, where encoding is performed in the unit of w-bit words (usually $w=8$). For EC(m,k), we have k parity stripes $p_j, j=1,2,\cdots,k$, for m original data stripes $d_i, i=1,2,\cdots,m$, and the $m+k$ stripes are called an EC group which ensures durability under any k failures. The parity stripes $p_j, j=1,2,\cdots,k$, is calculated by

$$(p_1,p_2,\cdots,p_k)^T = A\times(d_1,d_2,\cdots,d_m)^T, \qquad (1)$$

where $A=[a_{ij}]_{m\times k}$ is the encoding coefficient matrix.

Small writes are dominant for many real-world workloads in cloud storage systems, so it is important to efficiently support EC *partial writes*, i.e., writes on some part of an entire EC group. Early EC storage systems applies in-place update [6] to both data and parity, which leads to frequent disk seeks on hard-disk drives (HDDs).

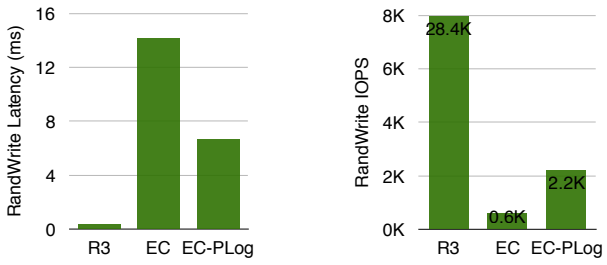


Figure 1: EC vs. replication in (cached) write latency and IOPS. R3: 3× replication with backup logging. EC: erasure coding with in-place update (no logging). EC-PLog: erasure coding with parity logging.

We compare EC (using in-place update) to replication in a small testbed of three machines, demonstrating EC suffers from poor write performance (Fig. 1).

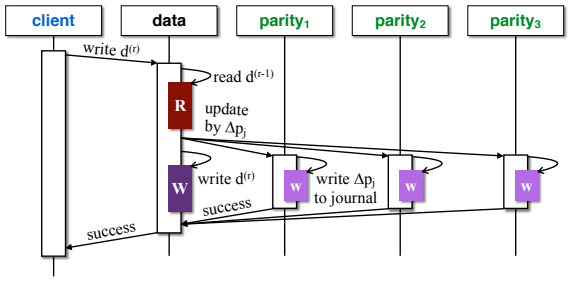
Log-based EC storage systems [16, 14, 8] improve small writes by appending the writes to a journal. Logging transforms random small writes into sequential writes to the journal, and thus (for HDDs) it avoids frequent disk seeks and boosts the write performance compared with in-place update. However, log-based approach suffers from poor read performance since the data is scattered in the journal. Parity logging (PL) [29] adopts a hybrid approach to alleviate this problem. It adopts in-place update to write the data and uses logging to write the parities. Since the parities will be read only when some data is unavailable, it improves small write performance without affecting normal reads.

State-of-the-art PL schemes [29, 17, 8] log the *parity delta* for each partial write, which will be used to "patch" the parity during journal replay. When updating a data stripe d_i, the delta Δp_j of parity stripe $p_j, j=1,2,\cdots,k$, is calculated by

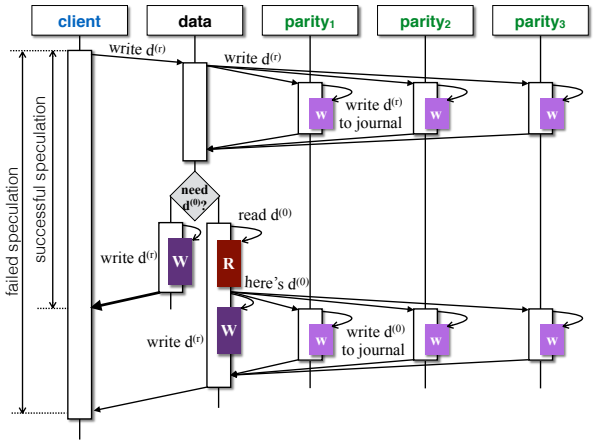
$$\Delta p_j = a_{ij}\times\Delta d_i, \qquad (2)$$

where Δd_i is the delta of data stripe d_i and $a_{ij}\in A$ is the encoding coefficient.

According to Eq. (2), for the rth write on a data stripe d_i (denoted as $d_i^{(r)}$), we first have to read $d_i^{(r-1)}$, the current value of d_i before this write, and we have $\Delta p_j = a_{ij}\times(d_i^{(r)}-d_i^{(r-1)}), j=1,2,\cdots,k$. Then we write the new data on the data server and send the k parity deltas to the parity servers. The entire procedure is illustrated in Fig. 2a. Our test shows that the latency of the write-after-read operation on 7,200 RPM HDDs is about 8.3 milliseconds, which is higher than that of pure write (due to one more disk seek). This contributes most to the performance degradation of partial writes in current PL schemes and results in significantly lower small write performance compared to replication (especially for cached writes), as shown in Fig. 1.



(a) Traditional partial writes in previous parity-logging read $d^{(r-1)}$ to compute Δp_j, which is appended to the parity journal.



(b) Speculative partial writes append $d^{(1)}$ and $d^{(0)}$ to the parity journal for the 1st write, and $d^{(r)}$ for the rth writes ($r \neq 1$), so as to avoid disk reads for *overwrites*.

Figure 2: Parity-logging vs. PARIX (in partial writes).

3 Speculative Partial Write

As discussed in §2, previous PL schemes have to perform disk reads for partial writes to compute the parity deltas, which will be used to update the parities during journal replay. Clearly, the key to improve the performance of partial writes is to reduce the number of reads.

Consider a series of r writes to the same data stripe d_i, say, $d_i^{(1)}, d_i^{(2)}, \cdots, d_i^{(r)}$, the parity stripes ($p_j, j = 1, 2, \cdots, k$) of which have not yet been replayed. Let $d_i^{(0)}$ and $p_j^{(0)}$ be the original values of the data d_i and parity p_j, respectively. By Eq. (2), we could update a parity stripe p_j by $\Delta p_j^{(1)} = a_{ij} \times \Delta d_i^{(1)}$, $\Delta p_j^{(2)} = a_{ij} \times \Delta d_i^{(2)}$, $\cdots$, $\Delta p_j^{(r)} = a_{ij} \times \Delta d_i^{(r)}$, and thus we have

$$p_j^{(r)} = p_j^{(0)} + \sum_{x=1}^{r} \Delta p_j^{(x)} = p_j^{(0)} + \sum_{x=1}^{r} a_{ij}(d_i^{(x)} - d_i^{(x-1)})$$
$$= p_j^{(0)} + a_{ij} \times (d_i^{(r)} - d_i^{(0)}). \quad (3)$$

By Eq. (3), the current parity stripes $p_j^{(r)}, j = 1, 2, \cdots, k$, could be calculated by $p_j^{(0)}$, $d_i^{(0)}$, and $d_i^{(r)}$.

This enables us not to use the delta of the parity, but to use the delta of the data itself, i.e., the difference between the data's latest and original values ($d^{(r)}$ and $d^{(0)}$, where for conciseness we omit the subscripts), to calculate the parities. Consequently, $d^{(0)}$ only needs to be read once when writing $d^{(1)}$. As shown in Fig. 2b, for each write the data server *speculatively* sends the latest value ($d^{(r)}$) to the parity servers without reading $d^{(0)}$. The data server reads $d^{(0)}$ only when the parity servers explicitly request it by returning an error code `NEED_DO`.

Note that in Fig. 2b the data server does not know whether $d^{(0)}$ is needed before receiving responses from parity servers. This is because $d^{(0)}$ is needed every time after the log gets merged into the parity chunk, which is performed independently by every parity server. It is too expensive to maintain the consensus about whether $d^{(0)}$ is needed for every chunk on every parity server, as it introduces overwhelming communication cost, memory footprint and design complexity.

For a series of r writes, $d^{(1)} \sim d^{(r)}$, the speculation will succeed for $r-1$ writes ($d^{(2)}, d^{(3)}, \cdots, d^{(r)}$) and will only fail once ($d^{(1)}$). Consequently, PARIX avoids disk reads (on the data server) for the last $r-1$ writes while only introducing a small penalty of an extra network RTT to the first one.

A partial write to an EC group performs both random writes to the *data* and sequential appends to the *parity*. Although PARIX and previous PL schemes have similar overhead in performing appends on the parity servers, PARIX remarkably outperforms them in performing writes on the data server: for a non-cached (resp. cached) *overwrite*, PARIX's overhead is a disk write (resp. a memory write), while previous PL schemes' overhead is a disk write after a disk read (resp. a memory write after a disk read) assuming the read is cache-missed.

If the speculation fails, $d^{(0)}$ needs to be sent from the data chunk to the parity chunk, introducing an extra network RTT of about $0.1 \sim 0.2$ milliseconds. The failed speculation also wastes extra network bandwidth, which is negligible for modern networks as the partial writes are small. It is the parity servers' responsibility to track whether $d^{(0)}$ is already in the log for its EC group.

Compared to existing PL techniques, the speculation-based scheme usually reduces the amount of reads and slightly increases the amount of writes when missing $d^{(0)}$. In the worst case (of large sequential one-shot writes), speculation might double the amount of writes. Besides, a few more extra bytes will be transferred between data/parity servers when missing $d^{(0)}$. Clearly, large sequential one-shot write workload is not suitable for the speculative partial write scheme, and could be recognized by an additional cache layer (which will be studied in our future work) and handled as full writes.

Full writes. Workloads in real-world applications per-

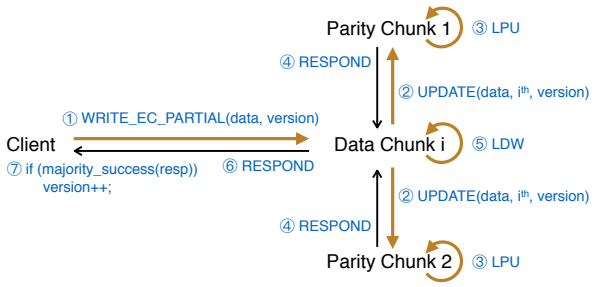


Figure 3: Partial writes with strong consistency guarantee. LDW: local data write. LPU: local parity update.

form not only random small writes but also large sequential writes, which induce *full writes* on the entire EC groups. For a full write $d = (d_i)$, $i = 1, 2, \cdots, m$, we first compute the parity $p = (p_j)$, $j = 1, 2, \cdots, k$ by Eq. (1) and write the parity into the corresponding parity servers. We then invalidate previous logs for the EC group in the journal by appending a special mark I. Therefore, the logs for data d on the parity journal are in the form of "$d^{(1)}$, $d^{(0)}$, $d^{(2)}$, $\cdots$, I, $d^{(1)}$, $d^{(0)}$, $d^{(2)}$, $\cdots$, I, $\cdots$". Note that $d^{(1)}$ is ahead of $d^{(0)}$ due to the speculation (Fig. 2b).

Replay. The replay of parity journals is asynchronously performed when the disk is idle. The (basic) replay procedure is straightforward. A process traverses the journal from the beginning, and for each parity *block* (the minimum unit of a disk sector) it records the original and latest data blocks ($d^{(0)}$ and $d^{(r)}$) in RAM. When encountering a mark I, it invalidates the records that are ahead of I. Finally it updates all the parities using the recorded original and latest values by Eq. (3).

4 PARIX Block Store

We have implemented a prototype of PARIX block store (PBS), which utilizes PARIX to provide virtual disks [18, 31, 21] that can be mounted by virtual machines (VMs) running cloud-oblivious POSIX applications. The design of PBS is similar to Blizzard [21] and URSA [1], except that PBS uses PARIX-backed EC (instead of *replication* in Blizzard and URSA) to achieve data durability.

PBS organizes its data and parity into fixed-size (normally 64MB) data/parity *chunks*. Like URSA [1], PBS leverages MySQL [4] and Redis [2] to implement a global master [9], which can be configured into the high-availability (HA) mode [33]. The master manages metadata [23] such as chunk ID/size and performance statistics, coordinates services like volume creation and recovery [19], and detects errors like missing servers and inconsistent chunks. Clients retrieve chunk information from the master, and read/write data through the chunk servers. PBS adopts no *nested striping* [21], because EC has essentially achieved the same effect.

Fig. 3 shows the partial write procedure in PBS. A client sends a write request to the data server, which forwards it to *all* relevant parity chunks on different parity servers. When receiving the write, the parity servers perform *local parity update* (LPU) to the *per-chunk* journal (Fig. 2b) and respond to the data server. Note that the data server cannot perform in-place *local data write* (LDW) for updating the data to its disk until this point, since if the parity servers request the initial value ($d^{(0)}$) in their response it will need to perform read-after-write (instead of pure write) and send $d^{(0)}$ to them.

PBS extends the basic replay procedure (§3). We maintain an index structure in RAM recording the positions of $d^{(0)}$ and $d^{(r)}$ for each parity block, so that in replay we could update a parity block by reading only the two blocks in the journal without traversing the journal. For EC(m,k), in the worst case the size of logs needed to be read from the journal for replaying a parity chunk is $2m$ times the parity chunk size, because calculating a parity block requires at most m data blocks each of which requires its own $d^{(0)}$ and $d^{(r)}$. Since the journal is replayed whenever the disk is idle, in practice its size is much smaller than $2m$ times the parity chunk size.

In the worst case the index structure keeps 2 addresses in RAM for each data block (of 512B), but the actual index size is much less than that, because: (i) the sizes of most small writes are at least 4KB (a page), instead of 512B (a block), which only requires to keep in RAM the first index and the size of each write; and (ii) a large write will immediately free all the in-RAM indices for the corresponding blocks. Assuming an average write size of 64KB, the in-RAM index size is at least three orders of magnitude smaller than the size of the data.

Recovery. When a data/parity chunk fails, the healthy data/parity blocks in the corresponding EC groups are read to perform the recovery. The unplayed parity logs in the journal are first replayed, similar to the aforementioned normal replay procedure. The small difference is that the recovery is pipelined: each parity block is used to calculate the failed block right after it is replayed.

Consistency. PBS uses a lease to ensure a virtual disk has at most one active client at any time and leverages (chunk-level) versioning [15] to guarantee per chunk strong consistency [11] (Fig. 3). The versioning mechanism is similar to that of parity logging [17], the details of which are omitted here due to lack of space.

5 Evaluation

This section presents evaluation results of the PBS prototype. Our testbed consists of 10 machines, each with dual 10-core Xeon E5-2630v4 2.20GHz CPU, 128GB RAM, one 10GbE NIC port, and 10 7200RPM HDDs. The machines connect to a non-blocking 10GbE network. The

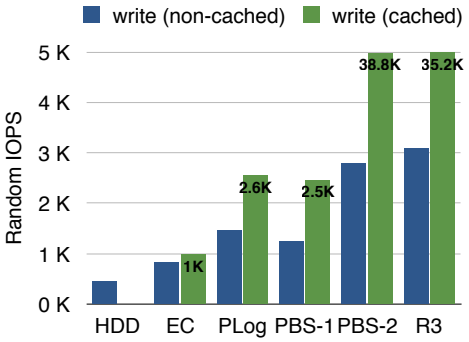


Figure 4: IOPS test. Background log flushing is omitted since PARIX only introduces a very small amount of extra data for logging compared to existing PL schemes.

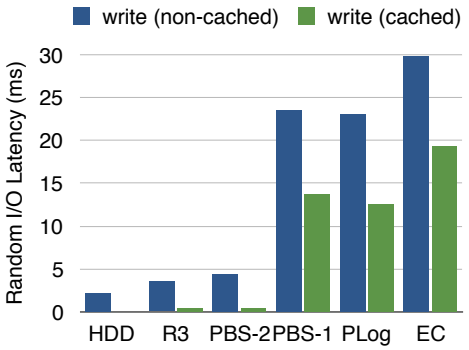


Figure 5: Latency test. Note that PARIX tries to eliminate unnecessary reads when performing speculative partial writes, so we measure random I/O latency so as to avoid the impact of prefetching and caching.

sizes of data/parity chunks and EC stripes are 64MB and 16KB, respectively. The virtual disk size is 100GB. The performance is measured by micro benchmarks, namely, small writes of 4KB block size (`fio --rw=randwrite bs=4KB`). §5.1 measures the performance of PBS in IOPS and latency, and §5.2 shows the recovery performance of PBS with different journal sizes.

5.1 PARIX Block Store

This section evaluates PBS. All measurements are performed on the VMs that mount virtual disks. For non-cached write, we turn off the cache in the OS and RAID cards, but keep the on-disk cache (otherwise the tests would not be able to get stable results). The queue depth is 1 and 32 for latency and IOPS tests, respectively.

Figs. 4 and 5 show the results in IOPS and random latency, respectively, where `HDD` represents the baseline performance of an HDD, `R3` uses 3× replication, `PBS-1` and `PBS-2` use EC(4,2) respectively with *failed* and *successful* speculation, `EC` is the standard EC(4,2) mode (no journal), and `PLog` uses traditional parity logging.

First, PARIX remarkably outperforms `PLog` when write-after-read is avoided in successful speculation. Second, PARIX is comparable to `PLog` even when the speculation fails, since the penalty is as small as an extra network RTT. Note that in order to compare different EC partial write schemes we must exclude the influence of read caching and prefetching, which exist in multiple layers in the I/O stack. Therefore, in Fig. 5 we measure the *random* (instead of *sequential*) I/O latency for all EC schemes, which may be up to more than 20 milliseconds unless the speculation succeeds (in `PBS-2`).

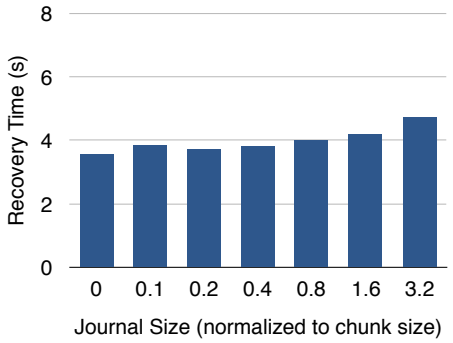


Figure 6: Single chunk failure recovery.

5.2 Recovery

We test the recovery performance of PARIX block store on 3 machines, using EC(4,2) with 64MB chunk size. A client first continuously performs small writes (of 4KB block size) until the (per parity chunk) journal size reaches a pre-defined proportion to the chunk size, which simulates the scenario that some corresponding parity logs in the journal have not yet been replayed before performing the recovery. We then emulate a data chunk failure by killing its service process. Fig 6 depicts the recovery times with respect to the exponentially-increased journal size (ranging from 0 to 3.2× chunk size). We do not test higher journal sizes, since in those cases replication would be even more efficient than EC and thus it might be inappropriate to apply PARIX. The result shows that the recovery overhead introduced by the parity journal is small, owing to the (in-RAM) full index.

6 Conclusion

This paper proposes PARIX for fast EC parity logging. We identify the root cause (write-after-read) for the poor performance of current EC partial writes, and speculatively performs pure write instead of write-after-read for small overwrites. We have implemented a prototype of PARIX block store (PBS). Evaluation shows that PBS remarkably outperforms current PL schemes. In the future, we plan to use PBS at the backend of our commercial block store in MOS (Meituan Open Service) [3].

Acknowledgement

This work is supported by the National Key Research and Development Program of China (2016YFB1000100) and the National Natural Science Foundation of China (61379055, 61379053, and 61222205). Yiming Zhang is the corresponding author. We thank Professor Ryan Huang, Huaimin Wang, Yijie Wang, Haibo Mi and Ziyang Li for their help to improve this paper.

References

[1] http://nicexlab.com/ursa/.

[2] http://redis.io/.

[3] https://mos.meituan.com/.

[4] https://www.mysql.com/.

[5] http://www.datacenterknowledge.com/archives/2008/05/30/failure-rates-in-google-data-centers/.

[6] AGUILERA, M. K., JANAKIRAMAN, R., AND XU, L. Using erasure codes efficiently for storage in a distributed system. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on* (2005), IEEE, pp. 336–345.

[7] CALDER, B., WANG, J., OGUS, A., NILAKANTAN, N., SKJOLSVOLD, A., MCKELVIE, S., XU, Y., SRIVASTAV, S., WU, J., SIMITCI, H., ET AL. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 143–157.

[8] CHAN, J. C., DING, Q., LEE, P. P., AND CHAN, H. H. Parity logging with reserved space: Towards efficient updates and recovery in erasure-coded clustered storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)* (2014), pp. 163–176.

[9] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. Bigtable: A distributed storage system for structured data. In *OSDI* (2006), pp. 205–218.

[10] CHEN, P. M., AND LEE, E. K. *Striping in a RAID level 5 disk array*, vol. 23. ACM, 1995.

[11] CHIDAMBARAM, V., SHARMA, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Consistency without ordering. In *USENIX FAST* (2012).

[12] 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), pp. 205–220.

[13] FORD, D., LABELLE, F., POPOVICI, F. I., STOKELY, M., TRUONG, V.-A., BARROSO, L., GRIMES, C., AND QUINLAN, S. Availability in globally distributed storage systems. In *OSDI* (2010), vol. 10, pp. 1–7.

[14] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *SOSP* (2003), pp. 29–43.

[15] GLENDENNING, L., BESCHASTNIKH, I., KRISHNAMURTHY, A., AND ANDERSON, T. E. Scalable consistency in scatter. In *SOSP* (2011), pp. 15–28.

[16] HUANG, C., SIMITCI, H., XU, Y., OGUS, A., CALDER, B., GOPALAN, P., LI, J., YEKHANIN, S., ET AL. Erasure coding in windows azure storage. In *Usenix annual technical conference* (2012), Boston, MA, pp. 15–26.

[17] JIN, C., FENG, D., JIANG, H., AND TIAN, L. Raid6l: A log-assisted raid6 storage architecture with improved write performance. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)* (2011), IEEE, pp. 1–6.

[18] LEE, E. K., AND THEKKATH, C. A. Petal: Distributed virtual disks. In *ACM SIGPLAN Notices* (1996), vol. 31, ACM, pp. 84–92.

[19] 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.

[20] 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 *SOSP* (1997), ACM.

[21] MICKENS, J., NIGHTINGALE, E. B., ELSON, J., GEHRING, D., FAN, B., KADAV, A., CHIDAMBARAM, V., KHAN, O., AND NAREDDY, K. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014), pp. 257–273.

[22] MURALIDHAR, S., LLOYD, W., ROY, S., HILL, C., LIN, E., LIU, W., PAN, S., SHANKAR, S., SIVAKUMAR, V., TANG, L., ET AL. f4: Facebook's warm blob storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), pp. 383–398.

[23] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J. K., AND ROSENBLUM, M. Fast crash recovery in ramcloud. In *SOSP* (2011), pp. 29–41.

[24] OUSTERHOUT, J. K., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., PARULKAR, G. M., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The case for ramclouds: scalable high-performance storage entirely in dram. *Operating Systems Review 43*, 4 (2009), 92–105.

[25] RIZZO, L. Effective erasure codes for reliable computer communication protocols. *ACM SIGCOMM computer communication review 27*, 2 (1997), 24–36.

[26] 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), 26–52.

[27] SATHIAMOORTHY, M., ASTERIS, M., PAPAILIOPOULOS, D., DIMAKIS, A. G., VADALI, R., CHEN, S., AND BORTHAKUR, D. Xoring elephants: Novel erasure codes for big data. In *Proceedings of the VLDB Endowment* (2013), vol. 6, VLDB Endowment, pp. 325–336.

[28] SELTZER, M., SMITH, K. A., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. File system logging versus clustering: A performance comparison. In *Proceedings of the USENIX 1995 Technical Conference Proceedings* (1995), USENIX Association, pp. 21–21.

[29] STODOLSKY, D., GIBSON, G., AND HOLLAND, M. Parity logging overcoming the small write problem in redundant disk arrays. In *ACM SIGARCH Computer Architecture News* (1993), vol. 21, ACM, pp. 64–75.

[30] THOMASIAN, A. Reconstruct versus read-modify writes in raid. *Information processing letters 93*, 4 (2005), 163–168.

[31] WANG, Y., KAPRITSOS, M., REN, Z., MAHAJAN, P., KIRUBANANDAM, J., ALVISI, L., AND DAHLIN, M. Robustness in the salus scalable block store. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (2013), pp. 357–370.

[32] WEATHERSPOON, H., AND KUBIATOWICZ, J. D. Erasure coding vs. replication: A quantitative comparison. In *International Workshop on Peer-to-Peer Systems* (2002), Springer, pp. 328–337.

[33] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), USENIX Association, pp. 307–320.

[34] ZHANG, Y., LI, D., GUO, C., WU, H., XIONG, Y., AND LU, X. Cubicring: Exploiting network proximity for distributed in-memory key-value store. *IEEE/ACM Transactions on Networking* (2017).

[35] ZHANG, Y., LI, D., TIAN, T., AND ZHONG, P. Cubex: Leveraging glocality of cube-based networks for ram-based key-value store. In *IEEE INFOCOM* (2017).

E-Team: Practical Energy Accounting for Multi-Core Systems

Till Smejkal[1], Marcus Hähnel[1], Thomas Ilsche[2], Michael Roitzsch[1], Wolfgang E. Nagel[2], and Hermann Härtig[1]

[1]Operating Systems Group,TU Dresden
[2]Center for Information Services and High Performance Computing (ZIH), TU Dresden
firstname.lastname@tu-dresden.de

Abstract

Energy-based billing as well as energy-efficient software require accurate knowledge of energy consumption. Model-based energy accounting and external measurement hardware are the main methods to obtain energy data, but cost and the need for frequent recalibration have impeded their large-scale adoption. Running Average Power Limit (RAPL) by Intel® enables non-intrusive, off-the-shelf energy monitoring, but only on a per-socket level. To enable apportioning of energy to individual applications we present E-Team, a non-intrusive, scheduler-based, easy-to-use energy-accounting mechanism. By leveraging RAPL, our method can be used on any Intel system built after 2011 without the need for external infrastructure, application modification, or model calibration. E-Team allows starting and stopping measurements at arbitrary points in time while maintaining a low performance overhead. E-Team provides high accuracy, compared to external instrumentation, with an error of less than 3.5 %.

1 Introduction

Energy has become the major factor constraining the utility of today's systems. For mobile platforms, which rely heavily on battery life, energy efficiency is an important differentiator for applications and devices. Being more energy efficient is a competitive advantage. In datacenters, energy is nowadays dominating the operation costs, necessitating energy-based payment models [21].

Accurate accounting of energy is paramount to optimize energy consumption and to enable energy-based billing. Software developers rely on energy consumption statistics to find and fix energy bugs [31] and improve the energy efficiency of their algorithms [18]. But software development already requires developers' attention to non-functional properties, like responsiveness and security. To enable energy efficient systems developers need a measurement infrastructure that is *easy to use* and *cost-effective* to deploy.

As energy characteristics often only manifest during runtime of the deployed application, such infrastructure must be *non-intrusive* in production environments by not incurring any performance loss or energy penalty when not in use. Still, enabling on-the-fly measurement of individual applications or parts of the system should be as easy as executing a simple command.

1.1 State of the Art

External measurement hardware is accurate [23], but can only provide machine-level measurements. Inference based solutions [35, 25, 7] are more flexible, but require calibration. We strive for a solution combining the respective advantages.

Intel introduced the Running Average Power Limit (RAPL) technology in Sandy Bridge™ CPUs [33]. It provides a power limiting infrastructure that is automatically calibrated during startup and exposes energy measurements. While not providing per-application energy values, it removes the need for expensive, specialized external measurement hardware and comes with zero setup effort. We introduce RAPL in Section 2.

Simple inference-based models, using CPU time or retired instructions, fail to accurately capture energy consumption of complex workloads making energy apportioning infeasible. We illustrate this point by measuring a busy loop that does not touch any data, and FIRESTARTER [14], a CPU burner application designed for high power-usage. Figure 1 shows the result of the experiment. We establish a baseline by measuring the energy consumption of each app in isolation using RAPL. Then we run both programs at the same time, scheduled by Linux' CFS, measure the system-level end-to-end en-

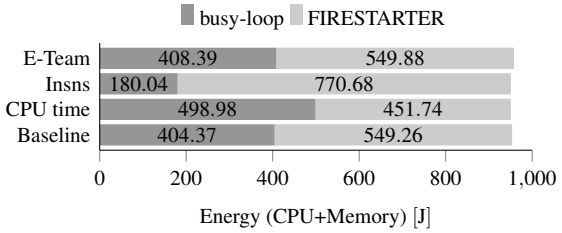


Figure 1: Energy attribution based on instructions retired (Insns) and CPU time, compared to E-Team

ergy consumption and apportion it based on CPU time and based on instructions retired. Both methods are incapable of correctly attributing energy consumption. We also give a short glimpse of the result of our solution, E-Team, which is able to accurately capture the energy consumption for both applications.

1.2 Contributions

We present the design and implementation of an operating system service for accurate and efficient measurement of per-application CPU energy use on multi-core systems using RAPL. Our key contributions are:

- A scheduler design to circumvent the limitations of RAPL using team scheduling (Section 3.1) and a Linux implementation (Section 4).
- A user-accessible interface to start and stop energy accounting of individual thread groups (Section 4.2).
- A scheduler integration of RAPL for short code paths (Section 2.3).
- An evaluation using standard benchmarks (NPB [1]) and real-world scenarios with multiple individually measured applications running in parallel (Section 5).
- A validation of our implementation's accuracy using a precise external measurement setup (Section 6).

2 RAPL for Energy Measurements

Starting with the Sandy Bridge generation, Intel CPUs provide the *Running Average Power Limit* technology (RAPL) [33]. As the name implies, RAPL is intended for power-limiting, but also provides energy counters. Due to the widespread availability and the fact that it requires no additional instrumentation, RAPL is used extensively for power and energy estimation [12, 16, 11, 39].

2.1 Basic RAPL Operation

RAPL provides energy measurements for four domains:

Package (PKG) the whole processor package,

Cores (PP0) aggregate of all cores in a package,

Graphics (PP1) the CPU-integrated graphics processing unit (not available on server platforms), and

Memory (DRAM) memory. Although officially only supported on server platforms [20], this domain is also available on desktop processors since Haswell.

The initial implementation of RAPL was based on a model using micro-architectural events to estimate energy consumption [8]. Hackenberg et al. [13] have revealed systematic errors in the RAPL energy counters, e.g. bias towards certain workloads and contradictory results when using Hyper-Threading. For Haswell generation processors, RAPL has been demonstrated to provide accurate measurements without systematic errors [15], hence the results presented in Section 5 and 6 were produced on Haswell desktop and server systems.

2.2 Limitations of Basic RAPL

Contrary to performance counters, RAPL counters are exposed exclusively through Model-Specific Registers (MSRs) that are only readable from kernel space. A number of methods exist in Linux to read the MSR in the kernel and make the values accessible to applications: Performance monitoring libraries such as PAPI [26] or LIKWID [38], and dedicated third-party drivers [24]. Since Linux 3.12, RAPL is usable as powercap driver. Since Linux 3.14 RAPL is accessible via the *perf* performance monitoring framework as system-wide performance-counter.

Another fundamental difference to conventional performance counters is that RAPL values are updated with an approximate frequency of 1 kHz [20] only, while performance counters are updated continuously. The update is not associated with a timestamp, preventing identification of stale values. The discrete updates make it difficult to measure code paths running shorter than or close to the counter's 1 ms update interval. The number of updates cannot be accurately determined: considering, for example, a piece of code running for 2.5 ms, it makes a large difference in terms of attributed energy whether there were two or three updates during that time.

This is especially visible in time-shared systems where switching between programs happens frequently. In these systems traditional performance counters are multiplexed by saving and restoring their values on every context switch. Such a technique cannot be trivially applied for RAPL because of the aforementioned update behavior. Counter values may be outdated at the point of context switching leading to significant measurement errors.

Similar to other measurement-based methods, mentioned in Section 1.1, the RAPL design cannot measure energy for individual cores or applications. Instead RAPL accounts the combined energy for all cores in a socket. This makes apportioning energy to an application executing in a multi-processor system with multiple,

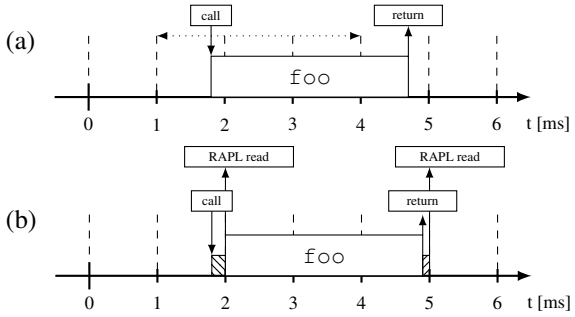


Figure 2: Synchronized measurement for short code

concurrently running applications non-trivial. In Section 3 we present the design of our scheduler-based measurement service. It ensures that, at any point in time, the cores of one socket are assigned exclusively to programs that should be measured together.

2.3 Measuring Short Code Paths

To address the problem of RAPL's fixed update intervals, we use a method from our previous work on measuring short code-paths [16]. When measuring a short code path, both the start and the end of the measurement may fall between the update points of the RAPL energy counter as illustrated in Figure 2 (a). The shorter the measured code path (here less than 3 ms) the higher the influence of measurement inaccuracies on the result. The measurement window, indicated by the dotted arrow, is offset against the code execution, delineated by *call* and *return*. The offset results in the inclusion of irrelevant code at the start and the omission of relevant code at the end. A solution to this problem is to synchronize the measurement time to counter updates. For the start of the measurement, this is achieved by repeatedly reading the `ENERGY_STATUS` register until it changes, indicating a RAPL update. Only then the measured code is executed.

Synchronizing the end of the function is not as simple. Just waiting for the next update will skew the measurement as the result would include the energy consumed while waiting. In our previous work, we propose to fill the time until the next update with a workload of fixed and known energy consumption [16]. Since polling the counter is needed to detect the update, using the polling loop as this defined workload elegantly solves the problem. Listing 1 shows pseudo-code for the algorithm executed when the function of interest terminates. The value of `ePerClock` is determined in a one-time calibration step performed by measuring the cost of repeatedly reading the RAPL counter over an interval of several updates (less than one second in total). Subtracting the known cost of the loop ensures that the returned value only contains energy consumed by measured code, thus effectively increasing the RAPL resolution and removing the constraint imposed by the update interval. Figure 2 (b) shows how this solution synchronizes the measurement with RAPL updates. The striped part before the measured code may have arbitrary energy consumption, the part at the end is the described polling loop. We call this mechanism *short-time RAPL*.

```
finish_measurement() {
    cycles = rdtscp();
    e_start = RAPLRead();
    while (RAPLRead() == e_start) {}
    cycles = rdtscp() - cycles;
    return RAPLRead() - ePerClock * cycles;
}
```

Listing 1: Algorithm for leaving measured code

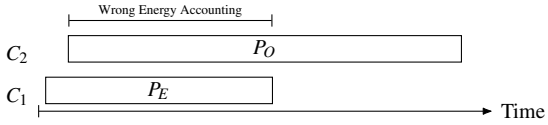


Figure 3: Miss-accounted energy in normal scheduling

2.4 Accounting Individual Processes

Figure 3 illustrates the challenge in accounting energy for individual processes using socket-local measurements. When using RAPL directly to measure process P_E running on core C_1, while an unrelated process P_O executes in parallel on core C_2, the final energy consumption will be incorrect. During the time marked in the graph the measurement will include the energy consumed by both programs. The example in Section 1.1 shows, that time-proportional accounting does not suffice to circumvent this problem as instruction types and access patterns are not taken into account. More sophisticated performance-counter-based models require extensive calibration leading to a significant deployment cost. Accordingly, we propose a different approach.

3 Energy Measurement as a Kernel Service

Our solution, E-Team, enables energy accounting on a per-application basis in multi-core systems through specialized scheduling. The main challenge of introducing a measurement infrastructure into a production system is to minimize the accompanying performance impact. The system should run as usual if there are no active measurements and hardware modifications should not be necessary. However, some overhead may be acceptable during troubleshooting or application development. Even in development scenarios isolating individual applications that are part of a complex system may be worthwhile to pinpoint energy bugs. Naïvely restricting the system to a single core or running the application of interest exclu-

sively in the system solves the energy accounting problem but does not resemble production system behavior.

For the remainder of this paper, we refer to individually executing programs as *process*es. Processes may be comprised of many execution contexts called *thread*s. Each thread is scheduled as a *task* by the scheduler and has an assigned task structure in the kernel.

We introduce the concept of *teams*. A team is an arbitrary group of tasks whose energy is accounted together. Teams get exclusive access to their assigned CPU socket to prevent tasks of different teams from running on the same socket in parallel. This enables the use of the socket-wide energy measurements of RAPL to account the team's energy consumption. Tasks of a team are scheduled on the team's socket according to any scheduling scheme. Thereby energy characteristics caused by interaction between measured tasks are largely preserved and performance degradation is limited.

We call this approach *team scheduling*. For the remainder of this paper we refer to a team that is measured as a *measured team*. There exists exactly one team containing all tasks that should not be measured (the *non-measured team*). To enforce the team-scheduling policy, we added a new scheduler to Linux.

3.1 Team Scheduling

The design of the E-Team scheduler guarantees that no tasks belonging to different teams run on the same socket at the same time. We want to enforce the following properties in our energy measurement service:

Property 1 (Team Interactivity)**:** *Teams are interruptible to enable interaction between different teams and maintain system responsiveness.*

Property 2 (Task Interactivity)**:** *Tasks of a team share the team's cores fairly to enable task interaction and preserve the team's energy characteristics.*

Property 3 (Accuracy)**:** *The scheduler limits switches between teams to curtail measurement errors due to multiplexing and uses short-time RAPL as required.*

Property 4 (Non-invasiveness)**:** *In the absence of measurements, the system behaves like an unmodified system.*

Property 5 (Usability)**:** *Starting and stopping measurements is possible at any point in time, either initiated by the user or the program itself. Teams grow and shrink when tasks are created, destroyed, added, or removed.*

To account energy for individual processes we propose to assign sockets exclusively to teams (measured teams or non-measured) in a time-multiplexed fashion. This leads to the main invariant of our scheduler:

Invariant 1: *On any socket only tasks of the same team can run concurrently at any point in time.*

Property 2 requires cores to be time-multiplexed between tasks of a team. The *team scheduler* controls which tasks can run on which socket at any given time based on policy and team-membership. A *task scheduler* distributes the tasks assigned to a socket between its cores. The team scheduler makes no assumptions about the task scheduler policy and the policy can be set per team. This allows to use the Completely Fair Scheduler (CFS) as task scheduler.

The team scheduler manages a list of teams (*team runqueue*), whereas each team consists of tasks called *team members*. One item in the team runqueue is the non-measured team. Teams are activated and deactivated by the team scheduler only as a whole. To activate a team, the team scheduler first deactivates the currently running team. It then dequeues the new team, affinitizes its tasks to the socket and notifies the responsible task scheduler to reschedule. Deactivation of a team entails removing all its tasks from the socket and adding the team back to the team runqueue. This design enforces Invariant 1. Only tasks of one team are available to the socket-local task scheduler to be scheduled.

When all tasks in the system belong to the same team, team scheduling is reduced to a no-op. This is the case when no measurements are taken as the non-measured team then contains all tasks. Together with the possibility to run arbitrary scheduling schemes within the task schedulers, this enforces Property 4.

While not implemented by us the extension of the proposed scheme to multiple sockets is straightforward. A running team can occupy multiple sockets at the same time. The team scheduler may remove or add sockets to a team as necessary. The maximum number of simultaneously active teams is limited by the number of sockets. Having multiple teams active on different sockets does not affect accounting accuracy, as each socket has its own RAPL domains.

The architecture of E-Team can be thought of as core-local scheduling with socket-level coordination.

3.2 Fine-Grained Context Switching

Property 3 is the hardest to enforce. Although the team scheduling approach guarantees that energy is only accounted for measured tasks, we still need to ensure that processes which execute for short times due to blocking are accounted accurately. To minimize overhead we try to switch teams only every 100 ms or more. This results in an error of about 1 % due to the fixed RAPL counter update intervals. The team scheduling frequency is tun-

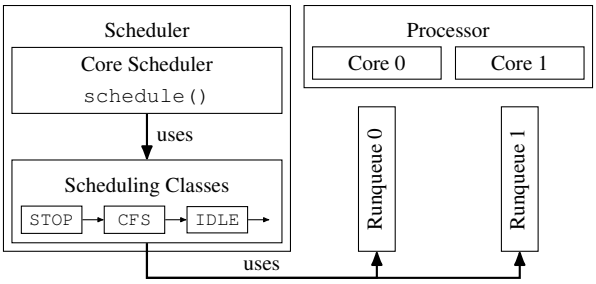


Figure 4: Linux scheduler architecture

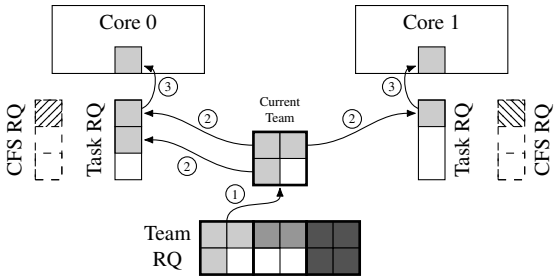


Figure 5: Scheduling a team for energy measurement

able, allowing the system operator to trade overhead against interactivity. More frequent team switches lead to higher system responsiveness at the expense of more overhead of the E-Team mechanism. Irrespective of the configured time-slice length, the measurement has to be stopped when all tasks within a measured team yield the CPU, which is regularly the case with I/O-bound workloads. To avoid measurement errors due to short executions, we employ the short-time RAPL technique introduced in Section 2.3 if less than 50 ms of the time slice are used. Otherwise we read the RAPL counters directly. This allows us to guarantee the accuracy property for all workloads, even interactive and I/O-heavy ones, while limiting the performance impact for compute-intensive workloads and execution phases.

4 Implementation

We implemented our energy measurement service E-Team as a scheduling class in the Linux kernel. The kernel patch and user tools are available on GitHub[1].

4.1 Scheduler Implementation

The general architecture of the Linux scheduling framework is illustrated in Figure 4. It consists of a core scheduler, which invokes the scheduling classes implementing the actual policies. Scheduling classes are sorted by priority. The highest priority is given to the *STOP* class, the lowest to the *IDLE* class. Schedulers maintain per-core runqueues, enabling them to make core-local scheduling decisions, which removes one of the bottlenecks in many-core systems. Usually, tasks in Linux are scheduled by the Completely Fair Scheduler (CFS). We prioritize the team scheduler above CFS.

The team scheduler must ensure that only tasks belonging to the same team are assigned to the same socket. The team scheduler maintains a list of teams (the team runqueue) where a team is a pointer to a list of the tasks that comprise the team. We illustrate the team scheduling process in Figure 5. As soon as the team scheduler decides — based on its team scheduling policy — to switch the team running on a socket it clears the core-local task runqueues . The team scheduler then picks the new team (step 1) and distributes its tasks to the cores of the socket (step 2) by enqueuing them in the task runqueues of the cores. CPU affinity is respected during this step. The core-local task schedulers are then triggered to reschedule the tasks in their task runqueue according to their scheduling policy (step 3). If there are not enough tasks in a team to occupy all cores of the assigned sockets, the idle task is scheduled on the remaining cores. This causes these cores to enter energy-saving states. When all the tasks in a team have terminated or the team's time slice is exhausted, the next team is scheduled.

Non-measured tasks are treated as an implicit team which is not managed by the team scheduler. In our implementation, they are not implemented as an actual team but as tasks kept in the core-local runqueues of CFS. These tasks are scheduled in between measured teams by yielding to CFS in order to enforce the interactivity property. How frequently E-Team yields to CFS depends on the number of non-measured tasks and the number of tasks in the measured teams, allowing fairness properties similar to CFS to be enforced.

Time-sliced round-robin with a base time-slice length of 100 ms was chosen as the team-scheduling policy. The base time-slice length is configurable. The actual length of the time slice depends on the number of ready tasks in the teams (i.e. the load). A team with more tasks waiting in its runqueues will get proportionally more time than a less-loaded team. This leads to fair multiplexing of CPU time between the team scheduler and the regular tasks in the system scheduled by CFS. We found a 100 ms base time-slice length to be a good compromise between overhead, accounting accuracy, and system-responsiveness. CFS chooses a similar base time-slice length for a system with CPU intensive load [6, Table 7-2]. Shortening the default time slice can improve responsiveness at the cost of higher overhead for frequent switching and more frequent use of the short-time RAPL mechanism. Please note that the default time-slice length is independent of the timer frequency. For all our experiments the timer

[1] https://github.com/TUD-OS

still ticked with a frequency of 1 kHz, thus invoking the scheduler every millisecond. The default time slice is a scheduler parameter that determines the *default* amount of time each process gets before it is rescheduled. The actual time may be less.

Tasks in the task runqueue are scheduled using time-sliced round-robin, but it would also be possible to use CFS as the task scheduling policy. Especially when measuring large teams containing many tasks with different priorities, CFS would better preserve the execution characteristics of the unmodified system.

When the E-Team scheduler does not schedule a measured team, it will yield to CFS, which then schedules the non-measured tasks as it normally would. This is an advantage of the implementation of the non-measured team using the normal CFS runqueues. Accordingly the system performs exactly as if it was unmodified whenever no tasks are in measured teams.

4.2 User-Level Tooling

Teams are formed by assigning a process to the E-Team scheduler. E-Team then automatically adds threads created by the process to the process's team. Although this simplified grouping was sufficient for our evaluation, it would also be possible to move individual threads to a team, disable the automatic addition of newly created threads, or combine several processes in one team.

The decision to use a specialized scheduler supports the usability property (see Section 3.1). Scheduler assignment is performed by starting the measured program through a tool such as `schedtool`. Alternatively our own tool `energy` can be used with the added benefit of outputting the energy consumption after program termination (like the Unix `time` utility does for time). Applications can start and stop measurement at arbitrary points in time by calling `sched_setscheduler` to move the process between the E-Team scheduler and CFS.

Applications can read their energy consumption from a file in their `procfs` subdirectory. Procfs provides runtime parameters and statistics of each process. We added two entries, `energystat` and `loopstat`, which provide access to the energy consumption and statistics about the scheduler's operation (number of time slices executed, short-time RAPL statistics, etc.), respectively. Listing 2 shows example output for the energy data.

Both files can be read during program execution to get regularly updated energy and statistics values. The content of the files will not change when the process is not measured and retains the values from when the process left the E-Team scheduling class. The files will retain their final values when the process stops being measured by leaving the E-Team scheduling class.

```
root@measure$ cat /proc/100/energystat
package (uJ)        : 1052792342
dram (uJ)           : 54277983
core (uJ)           : 842365434
gpu (uJ)            : 89234
updates             : 303
avg_loop_time (us) : 180
```

Listing 2: Example data provided by E-Team

5 Evaluation

To deliver on our promise of accurate energy accounting, we evaluate our system by first establishing a baseline using an unmodified system and then analyzing the energy and time overhead in three scenarios. We start by measuring a single application running alone on a Linux system. We then add background load and execute two applications in parallel, measuring them individually. Finally, we investigate the influence of short scheduling intervals and the effects of short-time RAPL.

Measurements were performed on a single-socket quad-core Intel® Haswell Core™ i7-4770 machine with 3.4 GHz nominal frequency and 2×4 GiB of DDR3 CL9 RAM clocked at 1333 MHz. We disabled Hyper-Threading and Turbo Boost, to make the individual measurements more deterministic and maintain comparability between single-application and multi-application runs. These options could otherwise lead to different behavior based on thread assignment and the decisions of Turbo Boost. We used a Linux 4.2.3 kernel in our experiments. Although we measured energy for all available RAPL domains, we only present PKG energy for brevity. The other domains showed comparable results.

5.1 Baseline and Overhead

Our first measurements establish the baseline for the rest of our evaluation. Baseline measurements were performed on a Linux system stripped down to the minimum necessary to run the benchmarks: We ran the system from an initrd with no system services interfering with execution. We believe the measured energy to conform to the energy consumed by the benchmarks. We use the NAS Parallel Benchmarks (NPB) [1], version 3.1, as benchmarks. Presented data is averaged over 20 consecutive runs. Error bars are not given in graphs if the standard deviation is below 1 %. Figure 6 shows the end-to-end measurement of the benchmarks for wall-clock time, cpu time and package energy (measured by the PKG counter). The benchmarks were scheduled using CFS. No parts of our kernel modification were active during the runs. Time was determined using the `time` command, while energy was measured by reading the RAPL MSR at the start and end of the benchmark. We measured each benchmark running with one to four threads.

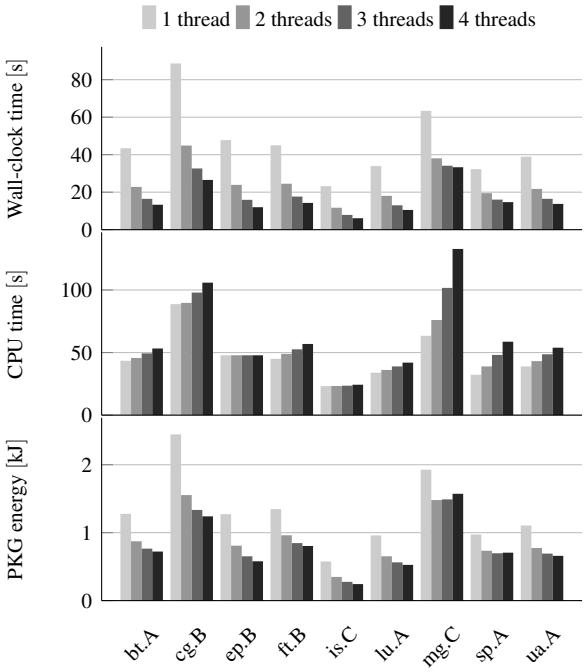


Figure 6: Baselines for wall-clock time, CPU time and PKG energy for different NPB kernels

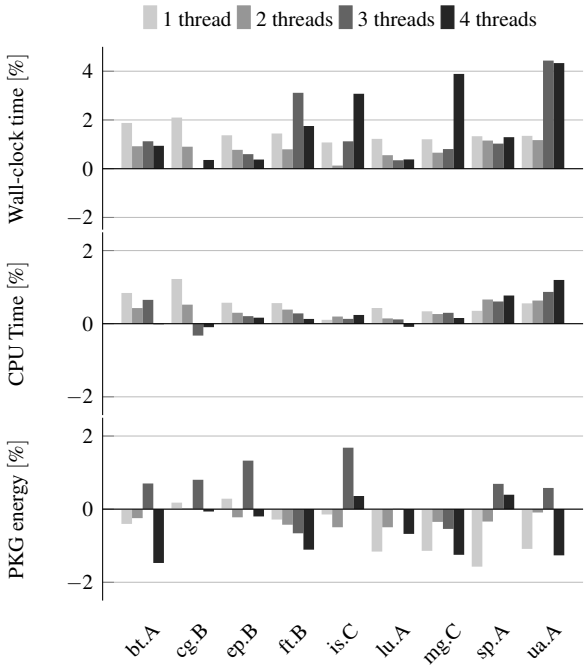


Figure 7: Wall-clock time, CPU time, and PKG energy measured using team scheduling compared to baseline.

We do not include the DC benchmark in our measurements because it mixes computation with extensive I/O. We found that its CPU time deviates significantly (>10 %) from wall-clock time when run as single application with one thread. The effect increases with the number of threads. This makes an end-to-end measurement meaningless as too much of the time is spent outside the benchmark. This is one of the cases that cannot be measured reliably without E-Team. We evaluate similar cases in Section 5.4 and will show a detailed discussion of DC in Section 6, when comparing against external measurements. For the other benchmarks, wall-clock time matched CPU time for the single-core case, resulting in a usable end-to-end baseline for energy.

Next, we repeated the baseline measurement using E-Team. This measurement and all the following in this section were performed on a normal Arch Linux system that was not stripped down. Ideally, the results obtained from E-Team would show the same wall-clock time and CPU time as the baseline. We also expected slightly lower energy consumption than the baseline, since E-Team does not account energy that is consumed by kernel tasks or by other processes in the system. Figure 7 shows the results of our measurements relative to the baseline. Team scheduling increases the wall-clock time of each benchmark. The more threads the program has, the longer it executes compared to the baseline, since background load in the system, even if single-threaded, blocks the whole measured program from running on the CPU. This is expected and the worst-case overhead is approximately 4 %. As we had hoped, CPU time did not increase significantly, which shows that the performance impact of our scheduler is negligible at less than 1 % in most cases. As long as there are enough tasks in all the teams, total performance of the system will not suffer. For package energy, our measurements are in the expected range with a difference relative to the baseline of less than 2 %. For most benchmarks we even measure less consumption due to the exclusion of unrelated work performed by the system. The measurements prove that our method combines low overhead with high precision.

5.2 Surveying Individual Groups of Tasks

After demonstrating that E-Team performs as good as the end-to-end measurements, we will show that our measurements stay accurate even in the presence of other tasks that are scheduled by the system. We introduce background load by running a single-threaded busy loop concurrently to the NPB suite. An empty busy loop does not touch any data and thus avoids any cache interference that could lead to changes in energy consumption.

Figure 8 shows the results of this experiment. We omit wall-clock time, as it is not a useful metric to compare against in this case. Wall-clock time will increase compared to the baseline in any case due to the intro-

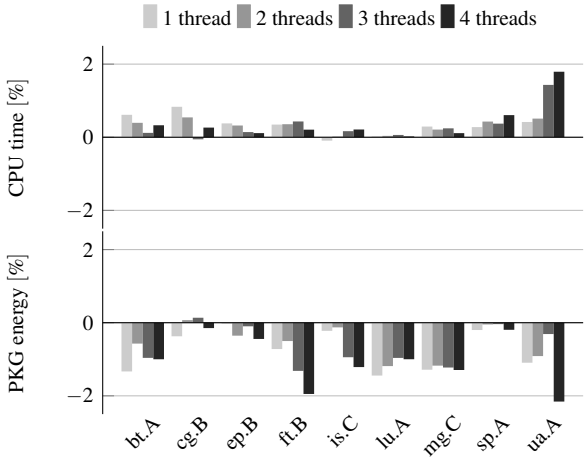


Figure 8: Per-application CPU time and PKG energy (relative to the baseline) as determined by E-Team with a background application running in parallel.

duced background load. For CPU time, we see an overhead of at most 2 %, while energy measurements are slightly below the baseline. We suspect the encountered energy reduction to be an artifact of precision limits of the RAPL counters. The busy-loop consumes significantly less energy than the benchmarks and we speculate that internal RAPL state influenced by this low-power activity bleeds into the results for the much more energy-consuming benchmarks. To test this hypothesis, we replaced the busy-loop with FIRESTARTER, which consumes more energy than the benchmarks. In this experiment, energy consumption increased relative to the baseline (e.g., 1.6 % for *ft.B*). This result indicates that inaccuracies within RAPL caused the measurement errors we observed. RAPL counters for DRAM proved less susceptible to this effect.

5.3 Multiple Measurements

One feature of our scheduler is that we can extract and measure a single application out of a number of applications running in parallel on the system. To demonstrate this feature we executed all application-pairs of the NPB suite (except for DC due to the lack of a meaningful baseline), measuring only one application of the pair. The results can be seen in Figure 9. We used scheduling slices of 100 ms to limit interference between the benchmarks. We ran this benchmark for one to four threads and compared the results against baseline.

As a guide to read Figure 9, consider the row with *ft.B* in the rightmost pane showing four threads in Figure 9b: Selecting the column *is.C* shows that the measured energy consumption of *ft.B*, when running concurrently with *is.C*, is 4 % below the baseline. No statement is made about *is.C* in this cell. Our worst-case error is 6 % for the DRAM energy (not shown) when running *ua.A* concurrently with itself. This may be attributed to either measurement errors introduced by our scheduler, errors in the RAPL model (i.e. incorrect energy values), or interference between the programs, despite the long scheduling interval. We will discuss the cause of this divergence in Section 6. Even a 6 % error is still on par with model-based estimation techniques [32, 37, 5].

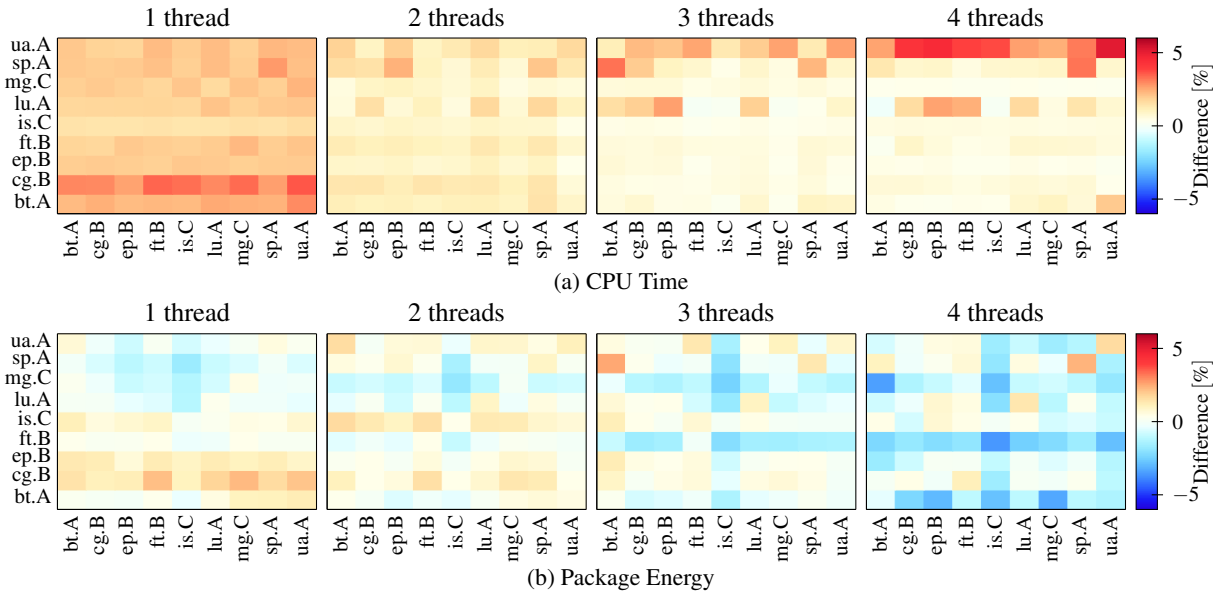
5.4 Short Scheduling Intervals

Particularly challenging for E-Team are applications that execute in short bursts, blocking in-between execution phases. Interactive GUI or multimedia applications as well as I/O-bound applications are examples that exhibit such behavior. They require rescheduling more often than our default time slice of 100 ms by yielding the CPU. Every time all the threads in the currently running team yield the CPU, we must switch to another team. If the last switch was not at least 50 ms ago, we need to perform short-time RAPL (refer to Section 2.3), to avoid inaccuracies introduced by the time-discrete updates of the RAPL counters. To evaluate the benefits of short-time RAPL for scheduling, we implemented a synthetic, interactive load that executes a busy loop for 5.4 ms, subsequently blocks for 1 s and then repeats the procedure 50 times. We compare short-time RAPL and naïve, update-oblivious multiplexing of the counter. As baseline we measure the busy loop that occupies the CPU as long as our synthetic workload (270 ms), but runs uninterrupted. Figure 10 shows the results. The energy measured by E-Team matches the baseline. When using naïve, update-oblivious multiplexing our measurements exhibit an error of up to 10 %. In contrast, short-time measurements only exhibit an error of 0.2 %. We conclude, that E-Team can reliably measure interactive and I/O-intensive tasks that yield the CPU frequently.

5.5 Practical Scenarios

Virtual machines We used `qemu-kvm` to run two VMs with Debian Jessie 8.4 64-bit, each given one core and 2 GiB of RAM. One VM was serving files over HTTP, the other was a malicious VM wasting CPU cycles by executing FIRESTARTER. We ran both VMs in parallel on Arch Linux using E-Team. Each VM received 300 s CPU time. The fileserver used 1034.1 J while the malicious VM used 7013.8 J. Based on this information a data-center operator could use appropriate billing or reduce the CPU time allocated to the malicious VM.

Single-Core Sampling We show the effectiveness of sampling to reduce overhead for single-threaded work-



(a) CPU Time

(b) Package Energy

Figure 9: Benchmarks in the rows are measured while running concurrently with the benchmarks in the columns. Shown is the relative difference to the baseline. We repeated this experiment for thread counts from one to four.

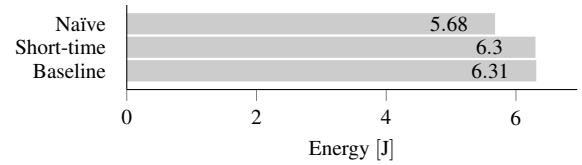


Figure 10: Short-time RAPL and update-oblivious measurements compared to baseline

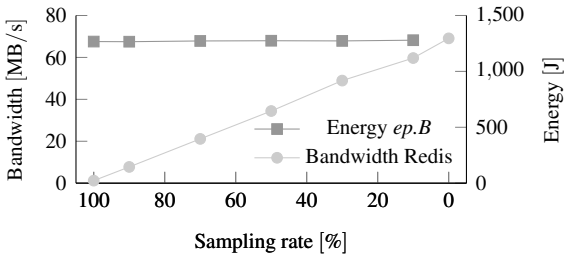


Figure 11: Redis throughput vs. *ep.B* measurement accuracy at various sampling rates

loads. Our setup consists of a single-threaded measured instance of *ep.B* running in parallel with unmeasured Redis, a popular in-memory database. To reduce the performance impact of isolating *ep.B* we employ random sampling. Figure 11 shows the throughput of memtier_benchmark at default configuration. At 10 % sampling we achive 86.4 % of the baseline performance of Redis, while maintaining 99 % measurement accuracy for *ep.B*. Like DC from NPB, Redis' I/O-intensive nature complicates determining an energy baseline. We thus omit its energy values.

6 External Validation

When running multiple teams in parallel, as done in Section 5.3, we do not know the cause of any aberrations from the baseline. Causes may be interference between threads, RAPL inaccuracies, or accounting errors in E-Team. To rule out the latter, we verify E-Team results using a secondary, external measurement infrastructure.

6.1 Measurement Setup

To verify the accuracy of our results, we use a sophisticated high-resolution power measurement infrastructure, which has been thoroughly verified [19]. It has been adapted to a Haswell-EP with two-socket Xeon E5-2690 v3 and a total of 256 GiB DDR4-2133 ECC RAM that we use as evaluation platform in this section.

We compare the results of RAPL against direct current (DC) measurements at inputs of each socket's voltage regulators. Both sockets are measured at a sampling rate of 500 kSa/s, to track power consumption between scheduling events. Data obtained from the external measurement infrastructure correspond to the sum of PKG and DRAM consumption according to RAPL. Because we measure at the input of the voltage regulators, the external measurements cover some components on the mainboard that are not measured by RAPL. Therefore RAPL reports less power consumption than the external measurements, even if both are perfectly accurate in their own power domain.

The verification is focused on identifying potential systematic inaccuracies introduced by our novel tech-

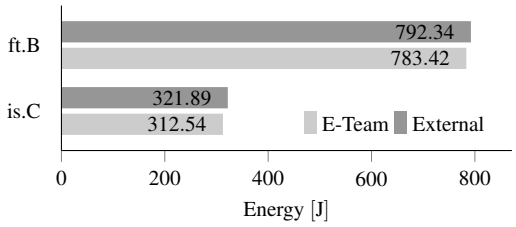


Figure 12: *ft.B* and *is.C* PKG energy using E-Team and external measurement with 12 threads per application

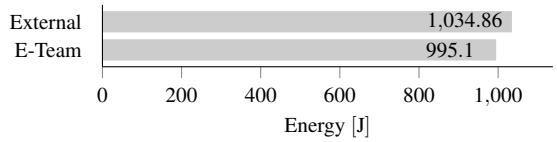


Figure 13: *dc.W* measured with E-Team and externally

nique. To compare the measured reference against the power domain of RAPL, we apply a model to map between the two. The model is trained on measurements of different workload kernels executed at various thread-counts and configurations as described in [4]. Training is performed on a non-modified Linux system using continuous RAPL and reference measurements. Linear regression provides the final slopes and intercepts separately for each socket with $R^2 > 0.999$.

Since the external measurement traces not only contain the power usage of the measured program but also of other tasks executed in parallel, a post-processing step was necessary to identify the regions in the traces during which the program of interest actually executed. For this purpose, we used an additional trace, generated by the E-Team scheduler, which indicates when each program was scheduled on the processor. We had to synchronize the traces, because they have timestamps from different clocks. We generated a special energy pattern before and after every measurement to correlate the measurement and scheduler traces.

6.2 Results

To validate our measurements from Section 5, we execute selected benchmarks on the instrumented hardware. We present the case of *ft.B* running together with *is.C*, which we already used in Section 5.3, as they exhibit significantly different power usage of 110 W and 80 W per socket, respectively. The results in Figure 12 show that our measurement is very accurate with an error of 1.1 % for *ft.B* and 2.9 % for *is.C*. The 12-core configuration used for the figure represents the worst case for this benchmark. The error decreased with fewer threads. We also examined the DC benchmark, which we were not able to evaluate in Section 5. We measured *dc.W* running with two threads and compared the external measurement to the E-Team result. Figure 13 shows that even for this I/O-intensive benchmark E-Team's error is only 3.5 %. Over 20 consecutive runs we observed a standard deviation well below 1 % in all cases.

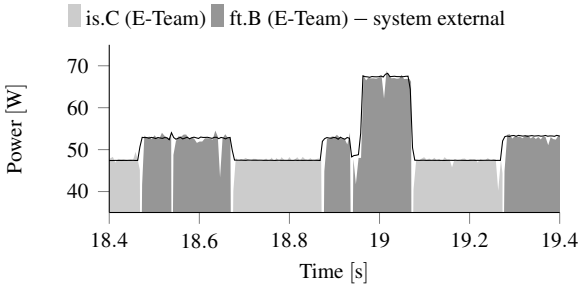


Figure 14: Power characteristics over time

Figure 14 shows that our E-Team implementation accurately tracks energy consumption over time. We ran *is.C* and *ft.B* in parallel, each in its own measured team and read their respective `procfs` entry repeatedly. We used a time slice of 200 ms. The characteristics of the external measurement match those of the internal one. The dips visible in the power consumption reported by E-Team (e.g. at 18.95 s) are caused by switches between teams or scheduling of the non-measured team. Tasks in the non-measured team yielded after very short time leading to short interruptions of the measured teams.

7 Limitations

E-Team provides accurate energy accounting for arbitrary groups of threads using socket-wide energy measurements. But this feature comes at a cost: a team always needs exclusive access to the socket. Accordingly, resources remain unused if teams cannot spread across all cores of the socket. The pathological example for this is a team that consists of a single thread. However, E-Team allows on-the-fly starting and stopping of measurements and thereby supports random sampling. This creates a trade-off space between measurement accuracy and performance overhead. In cases where the performance overhead of E-Team is prohibitive, limiting the measurement duration to short sampling intervals allows for acceptable performance at a slight loss of accuracy.

For I/O-bound workloads short-time RAPL is required more often, incurring additional overhead. In the worst case this translates to 1 ms overhead per team switch. Our experiments with *dc.W* showed only a performance degradation of 4.7 % in the worst case. We further examined a worst-case scenario for I/O-bound workloads by running `grep` recursively over the Linux source tree. We identified the extreme case showing 150 % overhead (155.32 s vs. 62.30 s) when flushing the buffer

cache before the run. We also measured Redis running memtier_benchmark and achived 30 % to 80 % of native performance for data sizes of 32 B to 128 kB despite its I/O-intensive nature. To measure such scenarios, we advice the use of random sampling.

8 Related Work

As energy efficiency is a cross-cutting concern, it has been approached from both the hardware and software side. On the hardware side, external measurement methods, such as those proposed by Hönig et al. [18], have improved significantly in sampling speed and accuracy over existing solutions, such as the frequently used Watts-Up power meter [9]. External measurements as data sources integrate well with our method, but introduce the need for additional hardware. Intel's RAPL addresses this problem by providing self-calibrating models [33]. Hackenberg et al. have shown that RAPL produces accurate energy estimates in recent versions [15] and compare various measurement methods [13].

Below the application layer, system architects construct runtimes and scheduling frameworks to model [30], account [29], and control [34] platform energy use. Several methods using performance counter based power models [22, 9, 36, 3, 2] exist. They exhibit relative errors in the range of 5 % to 10 % but can, contrary to RAPL, include other components such as disks. However, models require calibration, which has to be performed for each individual CPU. McCullough et al. found variations between individual CPUs of the exact same type to be too large to calibrate based on CPU model and have shown that linear CPU energy models are intrinsically limited in their accuracy [27].

There are various approaches using performance-counter-based models to apportion energy to VMs or applications. Shen et al. investigate Power Containers, which use model-based apportioning of energy to applications [36]. They use external recalibration during runtime, thus relying on additional hardware. Their methods exhibit relative errors of up to 11 % on Sandy Bridge CPUs. Bertran et al. account energy for VMs using a model-based approach and report 5 % relative error [3].

For high performance computing (HPC) systems, Georgio et al. have shown a SLURM-based job management system, which allows accounting of energy to jobs [12]. Their approach is limited to account energy on a per-node level. While suitable for typical HPC systems, it does not cover cloud or data-center scenarios with multiple simultaneous users per machine.

To schedule groups of tasks Ousterhout introduced co-scheduling [28] and an Feitelson et al. presented gang-scheduling [10]. Our method builds on these approaches.

9 Conclusion & Future Work

We presented the design and implementation of E-Team, a facility that enables accurate measurement of energy consumption for individual threads or groups of threads in a system. We isolate groups of interest using team scheduling. This enables us to use a system-wide measurement method, such as Intel's RAPL, while still being able to apportion energy consumption per thread or group of threads. To address the discrete nature of the RAPL readings, we employ short-time measurements to accommodate for applications that are interactive or yield the CPU often. We are able to isolate arbitrary parts of a system and apportion their energy with an error of at most 3.5 % compared to external measurements. Our methods provide greater accuracy than many existing model-based approaches and our validation shows that E-Team can apportion energy faithfully. To the best of our knowledge, our implementation is the first to allow practical, high-precision, per-application energy attribution in a multi-core system without relying on manual calibration or external measurement equipment.

Our implementation is applicable to a wide range of devices. E-Team does not rely on RAPL but can use other energy measurement techniques such as sensors available on mobile platforms [17] or hand-held devices.

Some ideas of our design are not yet implemented and are left for future work. We did not implement simultaneous execution of different teams on different sockets. The challenge in accounting energy on multiple sockets concurrently is that applications running on one socket can cause energy usage in another socket. Remote memory access is one example for such behavior. We leave the implementation of a cgroup-like interface to future work as well. Such an interface could prove useful to combine threads of multiple applications into one measured team. While we implemented random sampling, a detailed discussion of the performance and accuracy implications is left for future work, due to space constraints.

In summary, our work represents a significant step forward for data-center energy accounting, energy-based billing, and energy profiling of applications in production systems. E-Team provides a cheap, accurate, and easy-to-use solution for on-the-fly energy accounting.

Acknowledgements

This work is supported by the German Research Foundation (DFG) within the CRC 912 - HAEC. The authors would like to thank Mario Bielert for his work on the verification of the external measurement system.

References

[1] 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. S., ET AL. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications 5*, 3 (1991), 63–73.

[2] BASMADJIAN, R., AND DE MEER, H. Evaluating and modeling power consumption of multi-core processors. In *Future Energy Systems: Where Energy, Computing and Communication Meet (e-Energy), 2012 Third International Conference on* (2012), IEEE, pp. 1–10.

[3] BERTRAN, R., BECERRA, Y., CARRERA, D., BELTRAN, V., GONZALEZ, M., MARTORELL, X., TORRES, J., AND AYGUADE, E. Accurate energy accounting for shared virtualized environments using PMC-based power modeling techniques. In *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on* (2010), IEEE, pp. 1–8.

[4] BIELERT, M. Evaluating power estimation techniques: A methodological approach. Master's thesis, Technische Universität Dresden, 2016.

[5] BOSE, P., MARTONOSI, M., AND BROOKS, D. Modeling and analyzing CPU power and performance: Metrics, methods, and abstractions. *Tutorial, ACM SIGMETRICS* (2001).

[6] BOVET, D., AND CESATI, M. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[7] COLMANT, M., KURPICZ, M., FELBER, P., HUERTAS, L., ROUVOY, R., AND SOBE, A. Process-level power estimation in VM-based systems. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM, p. 14.

[8] DAVID, H., GORBATOV, E., HANEBUTTE, U. R., KHANNA, R., AND LE, C. RAPL: Memory power estimation and capping. In *Proceedings of the 2010 ACM/IEEE International Symposium on Low-Power Electronics and Design* (2010), ISLPED, IEEE, pp. 189–194.

[9] DO, T., RAWSHDEH, S., AND SHI, W. ptop: A process-level power profiling tool. In *Proceedings of the 2nd workshop on power aware computing and systems (HotPowerâĂŹ09)* (2009).

[10] FEITELSON, D. G., AND RUDOLPH, L. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing 16*, 4 (1992), 306–318.

[11] GAUTHAM, A., KORGAONKAR, K., SLPSK, P., BALACHANDRAN, S., AND VEEZHINATHAN, K. The implications of shared data synchronization techniques on multi-core energy efficiency. In *Presented as part of the 2012 Workshop on Power-Aware Computing and Systems* (2012).

[12] GEORGIOU, Y., CADEAU, T., GLESSER, D., AUBLE, D., JETTE, M., AND HAUTREUX, M. Energy accounting and control with SLURM resource and job management system. In *Distributed Computing and Networking*. Springer, 2014, pp. 96–118.

[13] HACKENBERG, D., ILSCHE, T., SCHÃŰNE, R., MOLKA, D., SCHMIDT, M., AND NAGEL, W. E. Power measurement techniques on standard compute nodes: A quantitative comparison. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on* (April 2013), pp. 194–204.

[14] HACKENBERG, D., OLDENBURG, R., MOLKA, D., AND SCHONE, R. Introducing FIRESTARTER: A processor stress test utility. In *Green Computing Conference (IGCC), 2013 International* (2013), IEEE, pp. 1–9.

[15] HACKENBERG, D., SCHONE, R., ILSCHE, T., MOLKA, D., SCHUCHART, J., AND GEYER, R. An energy efficiency feature survey of the Intel Haswell processor. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International* (2015), IEEE, pp. 896–904.

[16] HÄHNEL, M., DÖBEL, B., VÖLP, M., AND HÄRTIG, H. Measuring energy consumption for short code paths using RAPL. *SIGMETRICS Perform. Eval. Rev. 40*, 3 (Jan. 2012), 13–17.

[17] HÄHNEL, M., AND HÄRTIG, H. Heterogeneity by the numbers: A study of the ODROID XU+E big.LITTLE platform. In *6th Workshop on Power-Aware Computing and Systems (HotPower 14)* (2014).

[18] HÖNIG, T., JANKER, H., EIBEL, C., MIHELIC, O., AND KAPITZA, R. Proactive energy-aware programming with PEEK. In *2014 Conference on Timely Results in Operating Systems (TRIOS 14)* (2014).

[19] ILSCHE, T., HACKENBERG, D., GRAUL, S., SCHUCHART, J., AND SCHÖNE, R. Power measurements for compute nodes: Improving sampling rates, granularity and accuracy. In *2015 Sixth International Green and Sustainable Computing Conference (IGSC)* (Dec. 2015), The sixth international green and sustainable computing conference, pp. 1–8.

[20] INTEL. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A, 3B, and 3C: System Programming Guide*, Sep 2014. Section 14.9.

[21] JIMENEZ, V., GIOIOSA, R., CAZORLA, F. J., VALERO, M., KURSUN, E., ISCI, C., BUYUKTOSUNOGLU, A., AND BOSE, P. Energy-aware accounting and billing in large-scale computing facilities. *IEEE Micro*, 3 (2011), 60–71.

[22] KANSAL, A., ZHAO, F., LIU, J., KOTHARI, N., AND BHATTACHARYA, A. A. Virtual machine power metering and provisioning. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 39–50.

[23] KONSTANTAKOS, V., CHATZIGEORGIOU, A., NIKOLAIDIS, S., AND LAOPOULOS, T. Energy consumption estimation in embedded systems. *Instrumentation and Measurement, IEEE Transactions on 57*, 4 (2008), 797–804.

[24] Krapl: Intel RAPL driver exposing the RAPL interface in sysfs. `https://github.com/TUD-OS/krapl`.

[25] LI, T., AND JOHN, L. K. Run-time modeling and estimation of operating system power consumption. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2003), SIGMETRICS '03, ACM, pp. 160–171.

[26] MCCRAW, H., RALPH, J., DANALIS, A., AND DONGARRA, J. Power monitoring with PAPI for extreme scale architectures and dataflow-based programming models. In *Proceedsings of the 2014 IEEE International Conference on Cluster Computing* (2014), CLUSTER, IEEE, pp. 385–391.

[27] MCCULLOUGH, J. C., AGARWAL, Y., CHANDRASHEKAR, J., KUPPUSWAMY, S., SNOEREN, A. C., AND GUPTA, R. K. Evaluating the effectiveness of model-based power characterization. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2011), USENIXATC'11, USENIX Association, pp. 12–12.

[28] OUSTERHOUT, J. K. Scheduling techniques for concurrent systems. In *ICDCS* (1982), vol. 82, pp. 22–30.

[29] PATHAK, A., HU, Y. C., AND ZHANG, M. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 29–42.

[30] PATHAK, A., HU, Y. C., ZHANG, M., BAHL, P., AND WANG, Y.-M. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the 6th ACM European Conference on Computer Systems* (2011), EuroSys, ACM, pp. 153–168.

[31] PATHAK, A., JINDAL, A., HU, Y. C., AND MIDKIFF, S. P. What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services* (2012), MobiSys, ACM, pp. 267–280.

[32] RIVOIRE, S., RANGANATHAN, P., AND KOZYRAKIS, C. A comparison of high-level full-system power models. *HotPower 8* (2008), 3–3.

[33] ROTEM, E., NAVEH, A., ANANTHAKRISHNAN, A., RAJWAN, D., AND WEISSMANN, E. Power-management architecture of the Intel microarchitecture code-named Sandy Bridge. *IEEE Micro 32*, 2 (2012), 20–27.

[34] ROY, A., RUMBLE, S. M., STUTSMAN, R., LEVIS, P., MAZIÈRES, D., AND ZELDOVICH, N. Energy management in mobile devices with the Cinder operating system. In *Proceedings of the 6th ACM European Conference on Computer Systems* (2011), EuroSys, ACM, pp. 139–152.

[35] RYFFEL, S. LEA2P – The linux energy attribution and accounting platform. *Master's thesis, Swiss Federal Institute of Technology (ETH), Zurich, Switzerland* (2009).

[36] SHEN, K., SHRIRAMAN, A., DWARKADAS, S., ZHANG, X., AND CHEN, Z. Power containers: An OS facility for fine-grained power and energy management on multicore servers. 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. 65–76.

[37] SNOWDON, D. C., PETTERS, S. M., AND HEISER, G. Accurate on-line prediction of processor and memory energy usage under voltage scaling. In *Proceedings of the 7th ACM &Amp; IEEE International Conference on Embedded Software* (New York, NY, USA, 2007), EMSOFT '07, ACM, pp. 84–93.

[38] TREIBIG, J., HAGER, G., AND WELLEIN, G. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of the 39th International Conference on Parallel Processing Workshops* (2010), ICPPW, IEEE, pp. 207–216.

[39] WANG, W., CAVAZOS, J., AND PORTERFIELD, A. Energy autotuning using the polyhedral approach. In *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques, S. Rajopadhye and S. Verdoolaege, Eds., Vienna, Austria* (2014).

Scalable NUMA-aware Blocking Synchronization Primitives

Sanidhya Kashyap Changwoo Min Taesoo Kim
Georgia Institute of Technology

Abstract

Application scalability is a critical aspect to efficiently use NUMA machines with many cores. To achieve that, various techniques ranging from task placement to data sharding are used in practice. However, from the perspective of an operating system, these techniques often do not work as expected because various subsystems in the OS interact and share data structures among themselves, resulting in scalability bottlenecks. Although current OSes attempt to tackle this problem by introducing a wide range of synchronization primitives such as spinlock and mutex, the widely used synchronization mechanisms are not designed to handle both under- and over-subscribed scenarios in a scalable fashion. In particular, the current blocking synchronization primitives that are designed to address both scenarios are NUMA oblivious, meaning that they suffer from cache-line contention in an under-subscribed situation, and even worse, inherently spur long scheduler intervention, which leads to sub-optimal performance in an over-subscribed situation.

In this work, we present several design choices to implement scalable blocking synchronization primitives that can address both under- and over-subscribed scenarios. Such design decisions include memory-efficient NUMA-aware locks (favorable for deployment) and scheduling-aware, scalable parking and wake-up strategies. To validate our design choices, we implement two new blocking synchronization primitives, which are variants of mutex and read-write semaphore in the Linux kernel. Our evaluation shows that these locks can scale real-world applications by 1.2–1.6× and some of the file system operations up to 4.7× in both under- and over-subscribed scenarios. Moreover, they use 1.5–10× less memory than the state-of-the-art NUMA-aware locks on a 120-core machine.

1 Introduction

Over the last decade, microprocessor vendors have been pursuing the direction of bigger multi-core and multi-socket (NUMA) machines [16, 31] to provide large chunks of memory, which is accessible by multiple CPUs. Nowadays, these machines are a norm to further scale applications such as large in-memory databases (Microsoft SQL server [26]) and processing engines [34, 41]. Thus, achieving application scalability is critical for efficiently using these NUMA machines, which today can have up to 4096 hardware threads organized into sockets. To achieve high performance, various applications such as databases [26], processing engines [34, 41], and operating systems (OS) often rely on NUMA partitioning to mitigate the cost of remote memory access either by data or task placement. However, these approaches do not address how to efficiently modify shared data structures such as inodes, dentry cache, or even the structures of the memory allocator that span multiple sockets in a large multi-core machine. As a result, synchronization primitives are inevitably the basic building blocks for such multi-threaded applications and are critical in determining their scalability [2]. Hence, the state-of-the-art locks [4, 5, 10, 11, 23, 24], which are NUMA-aware, are the apt choice to efficiently exploit the NUMA behavior for achieving scalability on these multi-core machines.

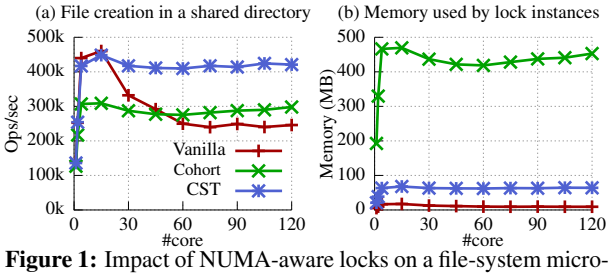


Figure 1: Impact of NUMA-aware locks on a file-system micro-benchmark that spawns processes to create new files in a shared directory (`MWCM` in [27]). It stresses either the `mutex` or the writer side of the read-write semaphore and memory allocation. Figure (a) presents the results up to 120 threads on a 120-core machine and Figure (b) shows the memory utilized by locks during the experiment. Here, *Vanilla* is Linux's native version, and Cohort is an in-kernel ported version of NUMA-aware locks [4, 11], and our NUMA-aware lock (CST).

NUMA-aware locks do improve application scalability, but they are difficult to adopt in practice. They either require application modification [5, 11, 23] or statically allocate a considerable amount of memory that can bloat shared data structures [4, 5, 11], as thousands to millions of lock instances can be instantiated in a large multi-core machine. For instance, a similar issue of adopting non-blocking queue-based locks occurred with Linux. Wickizier et al. [2] showed that a ticket lock suffers from cache-line contention with increasing core count. They replace it with the MCS lock to mitigate such an effect, which improved the system performance. Unfortunately, its adoption faced several challenges due to the change in the structure size and the lock function API [21].

We observe a similar trend in the case of blocking synchronization primitives, which suffer from numerous problems: 1) OS developers rely on TTAS locks or their variant [12, 18, 39], as they are simple and cache-line contention is not evident at smaller core count. However, they deter scalability on large multi-core machines (Figure 1 (a)). 2) The proposed blocking synchronization primitives [35, 36] are NUMA-oblivious and suffer from

high memory management cost for every lock acquisition, which impedes scalability. 3) NUMA-aware locks (Cohort locks) suffer from memory bloat as they statically allocate memory for all sockets, which is a serious issue in an OS [3] (Figure 1 (b)) and are non-blocking. 4) Finally, current blocking primitives severely suffer from the poor parking strategy because of cache-line contention, use of a global parking list, inefficient scheduling decisions, and inefficient system load estimation.

In this work, we design and implement two scalable blocking synchronization primitives, namely `CST-mutex` and `CST-rwsem`, from an OS perspective. Our primitives are memory-efficient, support blocking synchronization, and are tightly coupled with the scheduler, thereby resulting in better scalability beyond 100 physical cores for both under- and over-subscribed situations (tested up to 5× over-subscription). CST locks support blocking, as they incorporate a timeout capability for waiters, including readers and writers, in which waiters can park and wake-up without hurting the performance of the system. We use four key ideas to implement a scalable blocking synchronization primitive: First, we consciously allocate memory by maintaining a dynamic list of per-socket structures that is a basic building block of NUMA-aware locks. Second, instead of passing the lock to the very next waiter, we pass it to a not-yet-parked (still spinning) waiter, which removes the scheduler intervention while passing the lock to a waiter. Third, we keep track of the parked waiters in a separate, per-socket list without manipulating the actual waiting list maintained by the lock protocol. Lastly, we maintain a per-core scheduling information to efficiently estimate the system load. Thus, our blocking primitives improve the application performance by 1.2–1.6×, and they are 10× faster than existing blocking primitives in over-subscribed scenarios for various micro-benchmarks. Moreover, our approach uses 1.5–10× less memory compared with the state-of-the-art NUMA-aware locks.

In summary, we make the following contributions:

- **Two blocking synchronization primitives.** We design and implement two blocking synchronization primitives (`CST-mutex` and `CST-rwsem`) that efficiently scale beyond 100 physical cores.
- **Memory-efficient data structure.** We maintain a dynamically allocated list of per-socket structures that address the issue of memory bloat.
- **Scheduling-aware parking/wake-up strategy.** Our approach mitigates the scheduler interaction by passing the lock to a spinning waiter and batching the wake-up operation.
- **Lightweight schedule information.** We extend the scheduler to estimate the system load to efficiently handle both over- and under-subscription cases.

2 Background and Motivation

We first classify prior research directions into two categories: NUMA-aware locks and runtime contention management. We later give a primer on blocking synchronization primitives used in Linux.

NUMA-aware locks. NUMA-aware locks address the limitation of NUMA-oblivious locks [25] by amortizing the cost of accessing the remote memory. Most of the locks are hierarchical in nature such that they maintain multiple levels of lock [6, 10, 11, 14, 24] in the form of a tree. Inspired by prior hierarchical locks [10, 24], Cohort locks [6, 11] generalized the design of any two types of locks in a hierarchical fashion for two-level NUMA machines and later extended them for the read-write locks [4]. However, neither of them addresses the memory utilization issue nor supports blocking synchronization, which leads to sub-optimal performance when multiple instances of locks are used or when the system is overloaded. Besides Cohort locks, another category of locking mechanism is based on combining [13, 32] and the remote core execution approach [23] in which a thread executes several critical sections without any synchronization. Although it outperforms Cohort locks [23], the mechanism requires application modification, which is not practical for applications with a large code base.

Our design of NUMA-aware locks is memory conscious, as we defer the allocation of per-socket locks until required, unlike prior ones. In addition, CST locks are blocking, meaning that they support timeout capability while maintaining the locality awareness, unlike the existing NUMA-oblivious locks that allocate memory for each lock acquisition [35, 36]. Moreover, none of the NUMA-aware read-write locks support blocking readers, but the ones that do support [19, 28, 30] are NUMA oblivious and are designed specifically for read-mostly operations.

Contention management. The interaction between lock contention and thread scheduling determines application scalability, which is an important criterion to decide whether to spin or park a thread in an under- or over-subscribed scenario. Johnson et al. [17] addressed this problem by separating contention management and scheduling in the user space. They use admission control to handle the number of spinning threads by running a system-wide daemon that globally measures the load on the system. Similar approaches have been used by runtimes [7] and task placement strategies inside the kernel without considering the lock subsystem [42]. Along these lines, the Malthusian lock [9], a NUMA-oblivious lock, handles thread over-subscription by randomly moving a waiter from an active list to a passive list (concurrency culling), which is inspired by Johnson et al.

CST locks handle the over-subscription by maintaining a separate list in which waiters independently add them-

selves to a separate list after timing out. Our approach is different from the Malthusian lock and does not lengthen the unlock phase because wake-up and parking strategies are independent. Moreover, CST locks adopt the idea of a separate parking list from existing synchronization primitives [28, 29] or wait queues [38], but remove the cache-line bouncing by maintaining a per-socket, separate parking list for both readers and writers.

Design of Linux's `mutex` and `rwsem`. Many OSes, including Linux, do not allow nested critical sections for any blocking locks. The current design of `mutex` is based on the TTAS lock, which is coupled with a global queue-based instance [22] and a parking list per-lock instance. The algorithm works by first trying to atomically update the lock variable, called fast path; on failure, the mid-path phase (optimistic spinning) begins in which only a single waiter is queued up if there is no spinning waiter and optimistically spins until its schedule quota expires. If the waiter still does not acquire the lock, it goes to the slow-path phase in which it acquires a lock on the parking list (parking lock), adds itself, and schedules out after releasing the parking lock. During the unlock phase, the lock holder first resets the TTAS variable and wakes up a waiter from the parking list while holding the parking lock. Meanwhile, it is possible that either a new waiter can acquire the lock in the fast path or a spinning waiter in the mid path. Now, once a waiter is scheduled in, it again acquires the parking lock and tries to acquire the TTAS lock. If successful, it removes itself from the parking list and enters the critical section; otherwise, it schedules itself out again and sleeps until a lock holder wakes it up. The current algorithm is unfair because of the TTAS lock; even starves its waiters in the slow-path phase. Moreover, the algorithm also suffers from cache-line contention because of the TTAS lock and waiters maintenance, and even worse is the scheduling overhead in the slow-path phase and the unlock phase for parking and wake up.

The read-write semaphore is an extension of `mutex`, with a writer-preferred version. Both the write lock and the reader count are encoded in a word to decide readers, writer, and waiting readers. Moreover, `rwsem` maintains a single parking list in which both readers and writers are added. Thus, in addition to inheriting the issues of `mutex`, `rwsem` also suffers from reader starvation due to the writer-preferred version. Interestingly, developers found that the neutral algorithm suffers from scheduler overhead [20], while the writer-preferred version mitigated this overhead and improved the performance by 50% [37].

3 Challenges and Approaches

We present challenges and our approaches in designing practical synchronization primitives that can scale beyond 100 physical cores.

C1. NUMA awareness. A synchronization primitive should scale under high contention even in NUMA machines. Although locks that are used in practice [18, 28, 29] address cache-line contention by using queue-based locks [22] for high contention, they do not address the cache-line bouncing (remote socket access) introduced in NUMA machines. The remote access is at least 1.6× slower than the local access within a socket, which is a deterrent to the scalability of an application.

> **Approach:** To achieve scalability in NUMA machines, hierarchical locks (e.g., Cohort lock) are an apt choice. They mitigate the cache-line bouncing by passing a lock within a socket, which relaxes the strict fairness guarantee of FIFO locks for throughput.

C2. Memory-efficient data structures. Unfortunately, current hierarchical locks severely bloat the memory due to their large structure size (e.g., a Cohort lock requires 1,600 bytes in an eight-socket machine[1]), which they statically allocate for all sockets that may be unused. Memory bloat is a serious concern because it stresses the memory allocator and is alarming for synchronization primitives as they statically allocate the memory. For example, the size of the `XFS` inode structure increased by 4% after adding 16 bytes to the `rwsem` structure, which had an impact on the footprint and performance, as there can be millions of inodes cached on a system [8]. Thus, existing hierarchical locks are difficult to adopt in practice because they statically allocate per-socket structures during initialization.

> **Approach:** A hierarchical lock should dynamically allocate per-socket structure only when it is being used to avoid the memory bloat problem and reduce the memory pressure on a system.

C3. Effective contention management for both over- and under-subscribed scenarios. Designing synchronization primitives that perform equally well for both over- and under-subscribed situations is challenging. Non-blocking synchronization primitives, such as spinlocks including Cohort locks, work well when a system is under-loaded. However, for an over-loaded system, they perform poorly because spinning waiters and a lock holder contend each other, which deters the progress. On the other hand, blocking synchronization primitives such as `mutex` and `rwsem` are designed to handle an over-loaded system. Instead of spinning, waiting threads sleep until a lock holder wakes one up upon lock release. However, this procedure imposes the overhead of waking up in every unlock operation, which increases the length of the critical section. Also, frequent sleep and wake-up operations impose additional overhead on the scheduler, which can result in scalability collapse, especially when multiple lock instances are involved. To mitigate this issue, many blocking synchronization primitives [18, 28, 29] employ

[1] 64-byte cache line size × (3 cache lines for the socket lock × 8 sockets + 1 cache line for the top lock).

the *spin-then-park* strategy: a waiter spins for a while, and then parks itself out. Unfortunately, this approach is agnostic of system-wide contention, which leads to sub-optimal performance when multiple locks are contending. Ryan et al. [17] addressed the problem by designing a system-wide load controller, but its centralized design has memory hot spots for its control variables (e.g., the number of ever-slept threads) to decide whether a thread should sleep or spin.

Approach: To work equally well in both over- and under-loaded cases, we must address the system-wide load that allows waiters to optimistically spin in under-loaded cases and park themselves out in over-loaded cases. In addition, such a decision should be taken in a distributed way to keep the contention management from becoming a scalability bottleneck.

C4. Scalable parking and wake-up strategy. To implement an efficient blocking synchronization primitive, the most important aspects are how and when to park (schedule out) and wake up waiters with minimal overhead. The current approach [28, 29] maintains a global parking list to keep track of parked waiters and a lock holder wakes one of the parked waiters at the unlock operation. However, this design has several drawbacks: The frequent updating of a global parking list becomes a single point of contention in an over-loaded system, which leads to severe performance degradation because a lock holder has to wake up each sleeping waiter during the unlock phase, which adds extra pressure on the scheduler subsystem and lengthens the critical section: the cost of waking up varies from 2,000–8,000 cycles in the kernel-space or from 5,000–50,000 cycles in the user-space (`futex()` overhead). Thus, according to Amdahl's Law, an increased sequential part can significantly affect the scalability, especially in a large multi-core machine.

Approach: Instead of waking up the very next waiter, a lock holder passes the lock to a non-sleeping waiter, if any. Thus, this approach not only avoids waking up other threads under high contention, but also minimizes the access of the parking list and scheduler interactions. Furthermore, we maintain a per-socket parking list to remove costly cache-line bouncing among NUMA domains for accessing the parking list.

4 Design Principles

We present two scalable NUMA-aware blocking synchronization primitives, a mutex (`CST-mutex`) and a read-write semaphore (`CST-rwsem`), that can scale beyond 100 physical cores. At a high level, our lock is a two-level NUMA-aware lock, where a global lock is an MCS lock [25] and a per-socket local lock is a K42 lock [15] (see Figure 2). While the first level localizes the cache-line contention within a socket, the second one mitigates the cache-line bouncing among sockets. To enter a critical section, a thread first acquires the per-socket local lock and then the global lock. During the release phase, it first releases the global lock followed by the local lock. To mitigate memory bloating, we dynamically allocate the per-socket structure (`snode`) when a thread first tries to acquire the lock on a specific NUMA domain, and maintain it until the life-cycle of the lock. Each `snode` maintains a per-thread `qnode` in two lists: `waiting_list`—a K42-style list of waiters and `parking_list`—a list of parked (or sleeping) waiters. To acquire the lock, a thread first appends its `qnode` to the `waiting_list` of the corresponding `snode` in a `UW` (unparked waiting or spinning) status (`T3` in Figure 2) and spins until its schedule quota is over. On timing out, `T3` parks itself by changing its status to `PW` (parked waiting) and adds itself to the `parking_list` (`T2`). A lock holder (`T1`) that acquires its local and the global lock, passes the lock in the same NUMA domain by traversing the `waiting_list` during the release phase. It skips the parked waiter (`T2`) and passes the lock to an active waiter (`T3`). If there is no active waiter, the lock holder wakes up parked waiters in the same or other NUMA nodes to pass the lock. Our `rwsem` additionally maintains a separate reader parking list, besides writer parking list, to handle the over-subscription of the readers.

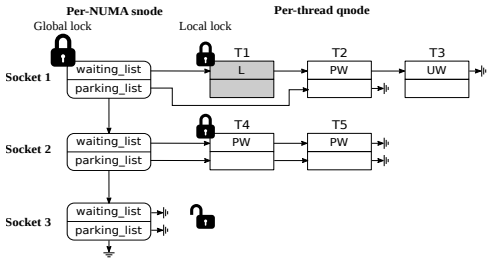


Figure 2: A `CST-mutex` is active on sockets: 1, 2, and 3. Currently, socket 1 is being served. `T1` now holds the lock (`L`); `T3` is spinning for its turn (`UW`: unparked waiting); `T2`, `T4`, and `T5` are sleeping (`PW`: parked waiting) until a lock holder wakes them up. A lock holder, `T1`, will pass the lock to `T3`, which is spinning, skipping the sleeping `T2`, to minimize the overhead of wake-up.

We explain our design principles on efficient memory usage (**C1** and **C2** in §4.1) and parking/wake-up strategy (**C3** and **C4** in §4.2). We later show how to apply our approaches to design blocking synchronization primitives: `CST-mutex` (§5.1) and `CST-rwsem` (§5.2).

4.1 Memory-efficient NUMA-aware Lock

Unlike other hierarchical locks that statically allocate per-NUMA structures for all sockets during the initialization, CST defers the `snode` allocation until the moment it is accessed first. The allocated `snodes` are active until the lock is destroyed. Our dynamic allocation of `snode` is especially beneficial in two cases: 1) when the number of objects is unbounded, such as `inode` and `mm_struct` in Linux kernel,[2] and 2) when threads are restricted to access

[2]The static allocation of all `snodes` increases the `inode` structure size by 3.8× and `mm_struct` size by 2.6× in an eight-socket machine.

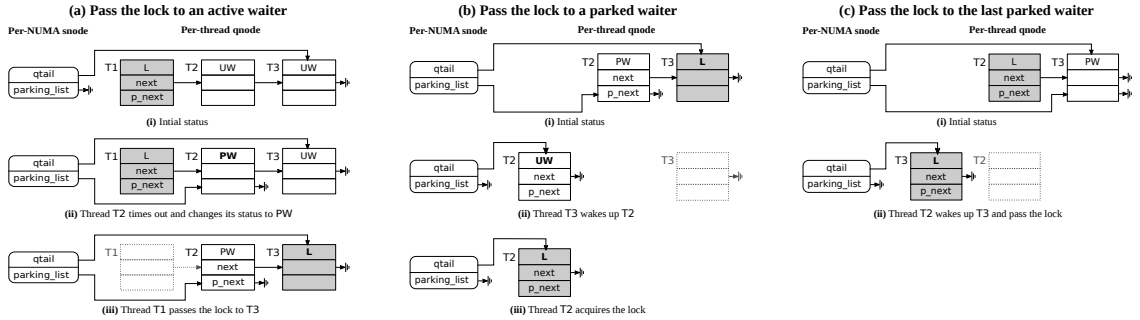


Figure 3: Figure (a) shows the passing of a lock to a spinning waiter inside a per-socket structure (`snode`). (i) `T1` is the current lock holder, and `T2` and `T3` are in the `waiting_list`, and `qtail` points to the `qnode` of `T3`. (ii) `T2` times out, successfully `CAS`es its state from `UW` to `PW`, and adds itself to the `parking_list`. (iii) `T1` exits the critical section. It tries to pass the lock to `T2`, but fails to `CAS` the state of `T2` from `UW` to `L`. `T1` goes to `T3` via `next` pointer of `T2`, successfully `CAS`es the state of `T3` from `UW` to `L`, and leaves the unlock phase. Figure (b) shows the passing of the lock to a parked waiter in the `parking_list`. (i) `T3` (lock holder) is in the unlock phase. It finds that `waiting_list` is empty as `T2` is in the `parking_list`. `T3` successfully `CAS`es `qtail` to `NULL`. (ii) Now, `T3` checks for parked waiters in `parking_list`, finds `T2`, and updates the state of `T2` from `PW` to `R`. (iii) Since `stail` is `NULL` and there are no prior waiters, `T2` sets its state to `L` and acquires the local lock, and later goes to acquire the global lock. Figure (c) illustrates the passing of the lock to a parked waiter at the end of the `waiting_list`. (i) On exiting the critical section, `T2` fails to `CAS` the state of `T3` to `L`, since it is parked. (ii) `T2` then explicitly `SWAP`s the state of `T3` to `L` and wakes it up. `T3` now holds the local lock and goes to acquire the global lock.

a subset of sockets such as running a multi-core virtual machine on a subset of sockets in a cloud environment.

For every lock operation, we first check whether a corresponding `snode` is present, and then get the `snode` to acquire the local lock. To efficiently determine whether an `snode` is present, a lock maintains a global bit vector in which each bit denotes the presence of a particular `snode`. Hence, each thread relies on the bit vector for determining the presence of an `snode`. We use `CAS` to atomically update the bit vector, but the number of `CAS` operations is bounded to the number of sockets in a system during the lifetime of a lock. A lock maintains allocated `snodes` in `snode_list`, which is traversed by a thread to find the corresponding `snode`. We separate the `snode` into two cache lines, almost-read-only for `snode` traversal and read-write for the local lock operation, which prevents `snode` traversal from incurring cache-line bouncing among sockets.

4.2 Scheduling-aware Parking/Wake-up Strategy

As discussed in the previous section, the most widely used *spin-then-park* policy fails to address the issue of scalability in NUMA machines. It works by maintaining a single, global parking list to account for the sleeping waiters, and wakes one or some waiters to pass the lock at the time of release. Hence, this approach is not scalable because it incurs contention on the parking list and suffers from scheduler interaction as it passes the lock to a potentially sleeping waiter in an over-subscribed condition.

To address these issues, the CST lock uses two key ideas: it maintains a per-socket `parking_list`, which minimizes costly cross-socket cache-line bouncing and passes the lock to a spinning waiter, whose time quota is not over yet, to minimize costly wake-up operations. We wake up a set of skipped sleeping waiters in bulk when there are no active waiters in the serving `snode` or pass the global lock to the other waiting `snode`. Thus, by relaxing the strict FIFO guarantee, we mitigate the lock-waiter preemption problem.

4.2.1 Low-contending List Management

In a CST lock, each `snode` maintains the K42-style waiting list that comprises its own tail pointer: `qtail`. For parked waiters, the `snode` also maintains a per-socket `parking_list` to account for the parked waiters, which avoids the costly cache-line bouncing while manipulating the `parking_list`. For a `rwsem`, we maintain a separate readers and writers `parking_list`, which simplifies the list processing in the unlock phase, as the lock holder can pass the lock to all parked readers or to one of the writers. Moreover, this approach enables a distributed parallel waking of readers at a socket level, which can improve the throughput of readers in an over-subscribed scenario (refer §5.2).

4.2.2 Scheduling-aware Parking/wake-up Decision

For a blocking synchronization primitive, the most important question is how to efficiently pass the lock or wake up a waiter, while maintaining an on-par performance in both the under- and over-subscribed cases. For the scalable parking/wake-up decision, we remove costly scheduler operations (i.e., wake-up) from the common, critical path and employ a distributed parking decision while considering the load on a system. We discuss three key ideas to address the problem of 1) whom to pass the lock to, 2) when to park oneself, and 3) how to take the parking decisions for blocking synchronization primitives.

Passing lock to an active spinning waiter. In queue-based locks (e.g., MCS, K42, and CLH), the successor of a lock holder always acquires the lock, which guarantees complete fairness, but, unfortunately, causes severe perfor-

mance degradation in an over-subscribed system, as this invariant stresses the scheduler to always issue a call to wake up the parked waiter. To mitigate this issue, we modify the invariant of a succeeding lock holder from the next waiter to a nearest *active* waiter, which is still spinning for the lock acquisition. Hence, the `waiting_list` comprises both active and parked waiters in its queue, and the parked waiters are added to a separate list: `parking_list`. Figure 3 (a) illustrates this scenario, where `T1` passes the lock to `T3` instead of `T2`, since `T2` is parked. Later, parked waiters are woken up in batches up to the number of physical cores in a socket once there is no active waiter in the `waiting_list`. When a parked waiter is woken up, it generally re-queues itself back at the end of the `waiting_list`, and again actively spins for the lock. This approach is effective because we can avoid scheduler intervention under high contention by passing the lock to an active waiter. In addition, a batched wake-up strategy amortizes the cost of the wake-up phase.

Scheduling-aware spinning. Current hierarchical locks [4, 6, 11] do not consider the amount of time a waiter should spin before parking itself out. Thus, in an over-loaded system, waiting threads and a lock holder will contend with each other, which deters the system progress. Instead, in CST locks, waiting threads park themselves as soon as their time quota is about to cease. To check the quota, we rely on the scheduler and its APIs for this information. Specifically in the Linux kernel, the scheduler exposes `need_resched()` to know whether the task should run, and preemption APIs (`preempt_disable() / preempt_enable()`) to explicitly disable or enable the task preemption. These APIs work with both preemptive and non-preemptive kernels. Limiting the duration of spinning up to the time quota proposed by the scheduler has several advantages: 1) It guarantees the forward progress of the system in an over-loaded system by allowing the current lock holder to do useful work while mitigating its preemption. 2) It allows other tasks to do some useful work rather than wasting the CPU cycles. 3) By only spinning for the specified duration, the primitive respects the fair scheduling decision of the scheduler.

Scheduling-aware parking. The current blocking synchronization primitives [28, 29] do not efficiently account for the system load; thus, they naively park waiters even in under-loaded scenarios. Hence, a naive use of the spin-then-park approach results in scheduler intervention, as the waiters park themselves as soon as their time quota ceases, and the lock holder has to do an extra operation of waking them up, which severely degrades the performance of the system in an under-loaded scenario [27]. Also, previous research [17] has shown that estimating system load is critical to the spin-then-park approach because it not only removes the scheduler interaction from the parking phase, but also improves the latency of the lock/unlock phase.

We gauge the system load by peeking at the number of running tasks on a CPU (i.e., the length of scheduling run queue for a CPU). Checking the number of running tasks is almost free because a modern OS kernel, including Linux, has a per-CPU scheduler queue, which already maintains an up-to-date per-CPU active task information. On the other hand, maintaining system-wide, central information, like the approach used by Johnson et al. [17], is costly because the cost of collecting the total number of active tasks increases with increasing core count, which may not catch the load imbalance due to the new incoming tasks or the rescheduling of periodic tasks.

5 Scalable Blocking Synchronizations

We now discuss the design and implementation of the two types of NUMA-aware blocking synchronization primitives (`mutex` and `rwsem`) using our design decisions. We first present the design of `mutex` (`CST-mutex`) along with the parking strategy and later extend it to `rwsem` (`CST-rwsem`). Figure 4 presents their pseudo-code.

5.1 Mutex (`CST-mutex`)

`CST-mutex` is a two-level hierarchical lock, which is extended to support blocking behavior by adding several design choices, such as scheduling-awareness, efficient spinning and parking strategy, and passing of the lock to the spinning waiter. The global lock employs an MCS lock, whereas the local lock is a K42 lock [15], a variant of the MCS lock. We choose the K42 lock because it does not require an extra argument in the function call as it maintains a `qnode` structure on the stack, but we can use any queue-based lock for the local lock. The top level lock maintains a dynamically allocated per-socket structure (`snode`) to keep track of the global lock and local lock information such as its `waiting_list` and the next waiter (for the K42 lock), and also `parking_list` information for the parked waiters. The MCS lock protocol has two status values: waiting (lock waiter) and locked state (lock holder). To support the blocking behavior, we keep the locked state (denoted as `L`) intact and extend the waiting state to the spinning/unparked (`UW`) and parked (`PW`) state. We also introduce a special state, called re-queue (`R`), that notifies the waiter to re-acquire the local lock.

Extended Cohort lock/unlock protocol. A thread starts by trying to acquire a local lock inside a socket. If there are no predecessors during the lock acquisition, it acquires the global lock, thereby becoming the lock holder, and enters the critical section (CS). The other threads that do not acquire the local lock are the local waiters, and the ones waiting for the global lock are the socket leaders. They wait for their respective predecessor to pass the lock. In the release phase, the lock holder locally tries to pass the lock to a successor. Thus, on success, the successor

```
1  def mutex_lock(lock):
2      snode = find_or_add_snode(lock) # Find or allocate snode once
3      while True:
4          lock_status = acquire_local_lock(snode)
5          if lock_status & ACQUIRE_GLOLBAL_LOCK is True: # Acquire global lock?
6              acquire_global_lock(lock, snode)
7              return
8
9  def acquire_local_lock(snode):
10     cur_qnode = init_qnode(status=UW, next=None) # Initialize qnode on the function stack
11     pred_qnode = SWAP(&snode.qtail, &cur_qnode) # Add to snode's waiting list
12     if pred_qnode is None: # Check for predecessor
13         cur_qnode.status = L|ACQUIRE_GLOLBAL_LOCK # Should acquire global lock
14         return cur_qnode.status
15     pred_qnode.next = &cur_qnode # Update predecessor next pointer
16     cur_qnode.task = current_task
17     while cur_qnode.status == UW: # Spinning for the local lock
18         if task_timed_out(cur_qnode.task): # Time quota is over
19             if park_write_qnode(snode, cur_qnode) == REQUEUE: # Check for requeue state
20                 if cur_qnode.status == L: # Local lock acquired
21                     break
22                 else:
23                     return R # Restart the local lock acquisition
24     update_next_qnode(snode, cur_qnode) # Update the next qnode (k42 protocol)
25     return cur_qnode.status
26
27 def acquire_global_lock(lock, snode):
28     snode = init_snode(snode, status=UW, next=None) # Initialize snode
29     pred_snode = SWAP(&lock.stail, &snode) # Add to global lock's waiting list
30     if pred_snode is None:
31         snode.status = L  # Acquired global lock
32         return
33     pred_snode.next_snode = snode # Update predecessor next pointer
34     snode.leader_task = current_task
35     while snode.status == UW: # Spin till the global lock holder passes the lock
36         if task_timed_out(current_task): # Leader time quota is over
37             if CAS(&snode.status, UW, PW): # Modify the state to PW
38                 schedule_out(snode.leader_task) # Schedule out the task
39     lock.current_serving_socket = snode
40
41 def mutex_unlock(lock):
42     snode = lock.current_serving_socket # Get the lock holder's snode #
43     if snode.local_batch_count < BATCH_COUNT: # local lock batching
44         snode.local_batch_count += 1
45         # Pass the lock to waiter with UW state and already has the global lock
46         if pass_local_lock(snode, acquire_global=False) is True:
47             return # Successfully found an active waiter
48     snode.local_batch_count = 0 # Reset the batch count
49     release_global_lock(lock, snode) # Release the global lock
50     release_local_lock(lock, snode) # Release the local lock
51     if snode_parking_list_is_not_empty(snode): # Remove parked waiter starvation
52         wake_up_parked_waiters(snode) # Wake up set of parked waiters
53
54 def release_local_lock(lock, snode):
55     if snode.qnext is None: # Check for next qnode, if any
56         if CAS(&snode.qtail, &snode.qnext, None) is True: # No qnode present
57             wake_up_parked_waiters(snode) # Wake up set of parked waiters
58         while snode.qnext is None: # qnode joined the qtail (waiting)
59             continue
60     if pass_local_lock(snode, acquire_global=True) is False:
61         with parking_list_lock(snode): # Acquire parking list lock to wake up a waiter
62             snode.qnext.status = L|ACQUIRE_GLOLBAL_LOCK # Update status
63             remove_from_parking_list(snode.qnext) # Update the parking list
64             schedule_in(snode.qnext.task) # Wake up the parked waiter
65
66 def release_global_lock(lock, snode):
67     if snode.next_snode is None: # Check for next snode, if any
68         if CAS(&lock.stail, snode, NULL) is True: # No snode present
69             return
70         while snode.next_snode is None: # Some snode joined the global lock stail
71             continue
72     if CAS(&snode.next_snode.status, UW, L) is False: # Check for parked snode
73         snode.next_snode.status = L # next snode is parked, still pass the lock
74         schedule_in(snode.next_snode.leader_task) # Wake it up for global lock acquisition
75 def park_write_qnode(snode, cur_qnode):
76     park_flag = False # Denotes whether waiter is parked
77     with parking_list_lock(snode): # Acquire parking list lock
78         if CAS(&cur_qnode.status, UW, PW) is True: # Try to update the state
79             add_to_parking_list(snode, cur_qnode) # Update parking list
80             park_flag = True # Parking was successful
81     if park_flag is True:
82         schedule_out(cur_qnode.task) # Schedule the task out
83         # cur_qnode.task is now awake, the task now returns REQUEUE
84         return REQUEUE # Should check for requeue phase
85     else:
86         return DO_NOT_REQUEUE # Acquired the lock
87
88 def pass_local_lock(snode, acquire_global):
89     qnode = snode.qnext # Search from snode.next
90     while True:  # Search for an active waiter
91         if CAS(&qnode.status, UW, L) is True:
92             if acquire_global is True: # Need to acquire the global lock
93                 L = L|ACQUIRE_GLOLBAL_LOCK # Update L status bit
94             return True
95         if qnode.next is None:
96             break
97         qnode = qnode.next # Find next qnode
98     snode.qnext = qnode # Found no one, updating qnext with tail
99     return False
100
101 def wake_up_parked_waiters(snode):
102     with parking_list_lock(snode): # Acquire the parking list
103         for qnode in parking_list(snode): # Iterate over stored parked waiters
104             qnode.status = R # All waiter should requeue right now
105             remove_from_parking_list(snode, qnode) # Update parking list
106             schedule_in(qnode.task)  # Schedule in the waiter
107
108 def write_lock(lock):
109     mutex_lock(lock)  # Acquire mutex first
110     for s in snode_list(lock):  # Check for active readers
111         while s.active_readers is not 0:
112             if task_timed_out(current_task):
113                 schedule() # Only schedule, will come back
114
115 def write_unlock(lock):
116     mutex_unlock(lock)  # Release the mutex
117     if lock.stail is None:  # There is no waiting snode
118         for s in snode_list(lock): # Traverse the snode
119             wake_up_first_read_waiter(s.reader_parking_list) # Wake-up a reader
120
121 def read_lock(lock):
122     snode = find_or_add_snode(lock) # Find or allocate the snode
123     ret = True
124     while True: # Spin, till acquired the lock
125         while lock.stail is not None: # Check for no waiters
126             if task_timed_out(current_task):
127                 ret = park_reader_task(lock, snode) # park the reader
128         if ret is True:
129             FAA(&snode.active_readers, 1)
130         if lock.stail is not None: # No one in the global lock tail
131             FAA(&snode.active_readers, -1)
132             ret = True
133             continue
134         break
135
136 def read_unlock(lock):
137     snode = find_or_add_snode(lock)
138     FAA(&snode.active_readers, -1) # Update the snode readers count.
139
140 def park_reader_task(lock, snode):
141     # Wait and park youself until global lock tail is NULL
142     park_and_wait_on_event(&snode.reader_parking_list, (lock.stail is not None))
143     if CAS(&snode.reader_is_parked_leader, False, True) is True:
144         FAA(&snode.active_readers, 1) # Decrease the active reader count
145         wake_up_all_read_waiters(&snode.reader_parking_list) # Wake up all readers
146         snode.reader_is_parked_leader = False
147         return False
148     return True
```

Figure 4: Pseudo-code of `CST-mutex` (lines 1 – 74), `CST-rwsem` (lines 108 – 148), and their parking/wake up (lines 75 – 106). We use three atomic instructions: `CAS(addr,new,old)` atomically updates the value at `addr` to `new` and returns `True` if the value at `addr` is `old`. Otherwise, it returns `False` without updating `addr`. `SWAP(addr,val)` atomically writes `val` to `addr` and returns the old value at `addr`. `FAA(addr,val)` atomically increases the value at `addr` by `val`.

does not acquire the global lock and immediately enters the critical section. To prevent starvation, a lock holder later passes the global lock to a globally waiting successor (socket leader) after a bounded number of local acquisitions. We now describe the `CST-mutex` protocol in detail, which is an extension of the aforementioned steps.

Acquire local lock: A thread `T` starts by first finding (or adding if not present) its `snode` (line 2). Unlike the Cohort lock protocol, `T` tries to acquire the local lock (line 4) in an infinite `for` loop because it may restart the protocol after being parked. In the local lock phase, `T` initializes its `qnode` (line 10) and then `SWAP`s the `qtail` of `snode` with `qnode`. It then acquires the global lock when no waiters are present. Otherwise, `T` spins on its status, which changes to either the `L` or `R` state (line 17). While waiting, `T` initiates the parking protocol (lines 75 – 86) on timing out, where it tries to `CAS` the status of `qnode` from `UW` to `PW`. `T` returns back on failure; otherwise, it adds itself to the `parking_list` and schedules out. Later, when a lock holder wakes it up, it resumes (line 83) and either acquires the local lock or restarts the protocol, depending on its updated status. If `T` has `L` status after being woken up, it goes on to acquire the global lock as the previous lock holder releases the global lock before waking up sleeping waiters. To mitigate cache-line bouncing, `T` checks for the global lock flag (line 5). If not set, `T` already holds the

global lock, or else it goes to acquire it.

Acquire global lock: T initializes its snode (line 28) and adds itself to the waiting_list (line 29). It then acquires the global lock if there is no waiter (line 31), or waits until its predecessor snode passes the lock (line 35). On timing out, while spinning (line 35), T CASes status of snode from UW to PW and schedules out (line 38); otherwise, it acquires the lock as the predecessor passed the lock. Note that even after being woken up, T always acquires the global lock without re-queueing itself.

Release local lock: T gets the current snode (line 42) and tries to locally pass the lock if it is within the batching threshold (line 43). To locally pass the lock, T first tries to CAS the status of its successor from UW to L. On success, the unlock phase is over; otherwise, it traverses the waiting_list to find an actively running waiter (lines 88 – 99). Figure 3 (a) illustrates this scenario, where T1 ends up passing the lock to T3 since T2 has PW state. Note that if all waiters are parked, (line 99), T releases the global lock (line 49) and then the local lock (line 50). T can also initiate both release phases when an snode exceeds the batching threshold. In the local unlock phase, T finds the snode qnext pointer to pass the lock. If qnext is NULL, T updates the qtail of snode with NULL (line 68) and wakes up waiters in the parking list to the R state to re-queue them back to the waiting_list (line 101). Figure 3 (b) illustrates the scenario, where T3 is the last one in the waiting_list. In the release phase, after resetting qtail to NULL, T3 wakes up parked T2 after updating its status from PW to UW. If there are waiters (lines 60 – 64), then T again tries to pass the lock to a spinning waiter in the waiting_list (line 88). If successful, a waiter acquires the local lock and then goes for the global lock since T has already released that one. If all, including the last waiter, are parked (lines 60–64), T passes the local lock to the last waiter and wakes it up because T cannot reset the qtail pointer, as there maybe some parked waiters; hence, passing the lock to the last waiter is mandatory. Figure 3 (c) shows this scenario in which T2 is about to release the local lock and finds that T3 is the last one and has PW status. T2 has to wake up T3 with an L state (not R), so that T3 can maintain the K42/MCS lock protocol.

Release global lock: The protocol differs from the MCS protocol for passing the lock. For an existing snode successor, thread T tries to CASes the status of its succeeding snode from UW to L. If successful, the lock is passed; otherwise, T explicitly updates the status to L and wakes up the succeeding socket leader (lines 72 – 74).

5.2 Read-write Semaphore (CST-rwsem)

CST-rwsem is a writer-preferred version of the Cohort read-write lock [4] (CST-rwsem) with two extensions: 1) application of our parking strategy to the readers and 2) our own version of the mutex algorithm (§5.1). It relaxes the condition of acquiring the CS by multiple threads in a *read mode*. Hence, it maintains an active reader count (active_readers) on *each* snode to localize the contention on each socket at the cost of increasing the latency for the writers. We further extend the snode to support the parking of readers by maintaining a separate parking list for them, which allows readers to separately park themselves without intervening with the writers.

Write lock: Thread T first acquires the CST-mutex (line 109). Then T traverses all snodes to check whether the value of active_readers is zero (line 111). Due to our writer-preferred algorithm, T blocks new readers from entering the CS because they can only proceed if there is no writer. Once the writer has acquired the mutex lock, it does not park itself, as this is a writer-preferred algorithm and the writer will soon enter the CS (lines 110 – 113).

Read lock: T first finds its snode (line 122) and waits until there are no writers (line 125). On timing out, while waiting, T adds itself to the parking_list and schedules itself out until there are no writers (line 142). The last writer wakes up the first reader in the parking_list, which wakes up remaining sleeping waiters in its own socket. Lines 143 – 146 present the waking up of the parked reader and subsequent readers.

Write unlock: T first releases the writer lock (line 116). If there are no writers (line 117), then T checks for any sleeping waiters across all snodes. If there are any, it wakes up the only very first waiter, which will subsequently wake up remaining waiters to acquire the read lock (line 119). This approach has two advantages: 1) it ensures distributed, parallel wake-up of the readers, and 2) it does not lengthen the writer unlock phase along with the least number of remote memory accesses.

Read unlock: Thread T searches for its snode from the list of existing sockets and atomically decreases the active_readers count by 1. T does not have to wake up any writer because our approach does not park the writer thread, which is going to be the next lock holder.

6 Implementation

We implemented CST locks on the Linux kernel v4.6 and v4.7. We also provide a destructor API to reclaim the snode memory while destroying a data structure (e.g., destroy_inode for inode). For our evaluation, we modified the inode structure to use our CST-rwsem in v4.7 and CST-mutex in v4.6 since mutex was replaced with rwsem from v4.7 [40]. We also modified the virtual memory subsystem (mmap_sem) that manipulates the virtual memory area of a process. We modified 650 and five calls for mmap_sem and inode, respectively. In total, our lock implementation comprises 1,100 lines of code and can substitute most of the lock instances in Linux.

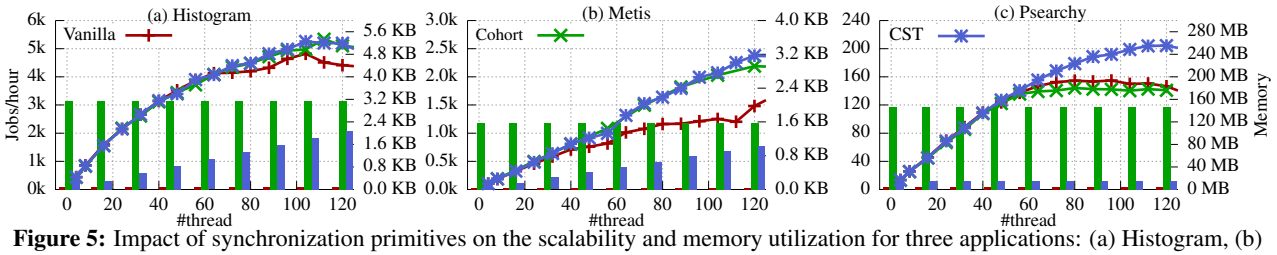


Figure 5: Impact of synchronization primitives on the scalability and memory utilization for three applications: (a) Histogram, (b) Metis, and (c) Psearchy with Linux's native `rwsem` (Vanilla), Cohort read-write lock, and `CST-rwsem`.

7 Evaluation

We evaluate the impact of CST locks by answering the following questions:

- How do locks affect the scalability and memory utilization of real-world applications? (§7.1)
- What is the impact of locks on operations provided by the OS in various scenarios? (§7.2)
- How does each design aspect help improve the performance? (§7.3, §7.4)

Evaluation setup. We evaluate CST locks on three workloads [1, 33] in an under-subscribed scenario, three micro-benchmarks from FxMark [27] that stress various file system components and the kernel memory allocator. Finally, we breakdown the performance implication of each design aspect using a hash table micro-benchmark. We evaluate on an eight-socket, 120-core machine with Intel Xeon E7-8870 v2 processors.

7.1 Application Benchmarks

We evaluate the scalability of `CST-rwsem` on three applications, namely Histogram [33], Metis [1], and Psearchy [1], that scale with increasing core count and stress the memory subsystem of the Linux kernel at varying levels. We compare our lock with the Linux's `rwsem` and an in-kernel port of Cohort locks [14]. For each benchmark results, we use *Vanilla* for the native Linux's `rwsem`, *Cohort* for the read-write Cohort lock, and CST for the `CST-rwsem` lock.

Histogram is a MapReduce application, which is page-fault intensive. It mmaps an 11 GB file at the beginning and keeps reading this file while each thread performs a simple computation. Figure 5 (a) shows that NUMA-aware Cohort and CST locks outperform the native implementation after 60 cores. They scale better because both locks localize the number of active readers within a socket, thereby having almost negligible contention across the sockets. Moreover, both locks have 2% idle time because the Cohort lock is non-blocking by design and the `CST-rwsem` effectively behaves as a non-blocking lock. On the other hand, the vanilla version is idle 10.5% of the time because of its ineffective parking strategy even in the under-subscribed situation. In summary, both locks outperform the native `rwsem` by 1.2× at 120 cores.

Metis is a mix of page-fault and mmap operation workload. It runs a worker thread on each core and mmaps 12 GB of anonymous memory for generating tables for map, reduce, and merge phases. Figure 5 (b) shows that both Cohort and CST locks outperform the original version by 1.6× as soon as the frequency of the write operation increases. Since the Cohort lock is non-blocking, it does not sleep, whereas the CST lock efficiently handles the under-subscribed case by not parking the threads, resulting in only 0.5% of idle time. Moreover, both locks batch readers, which improves the throughput of the workload. On the other hand, the original `rwsem` has 39% of the idle time because of its naive parking strategy and is 1.6× slower than the others at 120 cores.

Psearchy is a parallel version of searchy that does text indexing. It is mmap intensive, which stresses the memory subsystem with multiple userspace threads. It does around 96,000 small and large mmap/munmap operations from 96,000 files with multiple threads, which taxes the writer side of the `rwsem` in the memory subsystem as well as the allocation of the inodes for those files in the virtual file system layer. Figure 5 (c) shows that `CST-rwsem` outperform both the Cohort and native locks by 1.4× at 120 cores. Cohort locks suffer from the static allocation because the kernel has to allocate 96,000 inodes for reading files into a per-core hash table of Psearchy, which not only stresses the memory allocator with large objects, but also suffers from ineffective scheduling because of the involvement of multiple instances of locks. Like prior workloads, the native lock suffers from the scheduler intervention after 45 cores, as it spends up 54.4% being idle, whereas the `CST-rwsem` is only idle for 11.4% of the time.

Summary. Figure 5 shows the impact of scheduler intervention with increasing contention between readers and writers. With our efficient spinning strategy that checks its local load, CST locks have the same benefit as Cohort locks in the case of a highly contended but under-subscribed system. While Cohort locks improve the scalability of applications in highly contended and under-subscribed scenario, they hamper the scalability of applications that allocate multiple instances of locks (Figure 5 (c)). Unlike Cohort locks, CST locks consciously allocate memory with increasing *socket* count, which saves up to 10× of memory for each workload on a single socket, and 1.5 – 9.1× at 120 cores. Thus, CST locks show that dynamic allocation is beneficial to real applications, while mitigating the memory bloat issue and maintaining an on-par performance.

7.2 Over- And Under-subscribed Cases

We compare the performance of CST locks with the kernel and Cohort locks in both an over- and under-subscribed

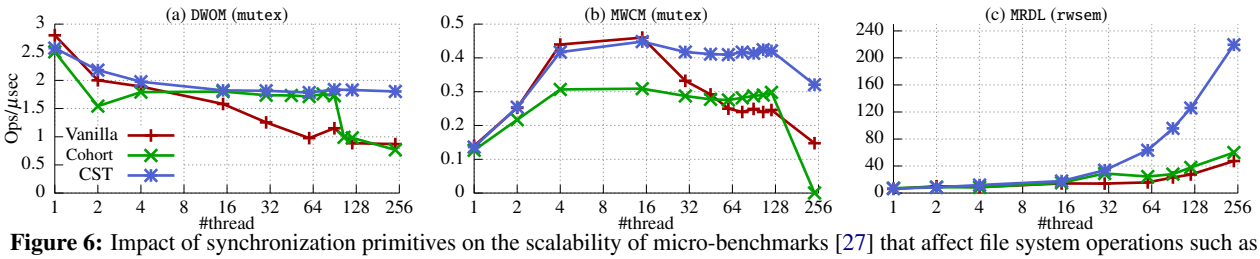


Figure 6: Impact of synchronization primitives on the scalability of micro-benchmarks [27] that affect file system operations such as (a) overwriting a block in a shared file, (b) creating, and (c) enumerating files in a shared directory.

system, where multiple instances of locks are in use. We run FxMark because it stresses various file system operations by only stressing various kernel components that interact with the virtual file system layer, without any user-space computation. We use three micro-benchmarks from FxMark [27] to show how multiple instances—static lock size allocation, contention, and scheduler—affect the scalability of file system operations: `DWOM` updates a shared file in which threads overwrite a block. It represents a log in I/O workloads such as databases that multiple threads share and manipulate. `MWCM` creates multiple files in a shared directory. Both stress the writer lock of `rwsem`, and `mutex`. Finally, `MRDM` enumerates all files in a shared directory and stresses the reader lock.

Block overwrite. Figure 6 (a) shows the impact of various locks on the scalability of block overwriting that stresses the `mutex`. We observe that the CST lock outperforms the Cohort lock by 1.6× and 2.3×, and the Linux one by 2.6× and 2.5× for 120 and 240 threads, respectively. Its efficient parking design maintains an on-par performance even in the over-subscribed scenario (i.e., 2× more threads). Cohort locks suffer from scheduler interaction because tasks get frequently rescheduled, which consume 54.4% of the time because of no scheduling information. The native `mutex` suffers from cache-line bouncing until 60 cores, but starts to suffer from scheduler intervention since the threads start parking themselves as the system is 98% and 90% idle at 120 and 240 threads, respectively.

File creation. Figure 6 (b) shows the impact of various locks on file creation. `CST-mutex` outperforms the Cohort lock by 1.4× and 1614.7×, and the Linux `mutex` by 1.7× and 2.2× for 120 and 240 threads, respectively. At 240 cores, `CST-mutex` suffers from a bottleneck imposed on the memory allocator because of the over-subscription, which also happens with the Linux `mutex`. The Cohort lock, stresses both the scheduler and the memory allocator, as each operation allocates a new inode, whose size is 3.8× larger than the normal inode structure. Moreover, at 240 cores, its performance severely degrades because of its non-blocking nature, and is 743.0× slower than the Linux `mutex`. The Linux version again suffers from the cache-line contention after 30 cores and then from scheduler intervention after 60 cores.

File enumeration. Figure 6 (c) shows the impact of reading a directory. `CST-rwsem` achieves almost linear scalability with increasing threads up to 120 cores and further scales in the over-subscribed case. It outperforms the Cohort lock by 3.3× and 3.7×, and the Linux one by 4.6× and 4.7× for 120 and 240 threads, respectively. The Cohort lock still suffers from scheduler interaction, whereas the Linux version suffers from cache-line contention because of the global count of readers compared with the per-socket storage by both hierarchical locks.

7.3 Performance Breakdown

We evaluate how each component of CST contributes to the overall performance improvement by using an in-kernel hash table that is protected by a single lock. To quantify the impacts of NUMA awareness and parking strategy, we keep the read-write ratio at 90/10%. We vary the thread count from 1 to 600 threads on 120 cores to show the effectiveness of our blocking strategy even in the over-subscribed scenario. Figure 7 (a) shows the throughput of readers with increasing thread count. We evaluate three variants of the reader-side parking strategy: 1) global wake-up of parked readers (CST-Wake) and 2) distributed wake-up (CST-DWake). In an under-subscribed system, CST variants outperform both Cohort and Linux by 4.6× and 10×, respectively, as Cohort locks suffers from scheduler intervention (86.4%) and `mutex` is contending on the global reader count value. Beyond 120 threads, both the Cohort and CST-Spin approaches perform poorly compared with Linux because they are non-blocking. On the other hand, CST-Wake and CST-DWake scale up to 600 threads, thereby showing the importance of blocking behavior. CST-DWake, a distributed wake-up scheme for readers, wakes up more readers in parallel, thereby improving their performance by 1.2× over the global wake-up strategy and outperforming the Linux version by 9.1×.

Figure 7 (b) presents another micro-benchmark results in which we update a single cache line by multiple threads from 120 to 600. We compare the Linux's `mutex` with the Cohort lock and two CST locks: 1) CST-WA is the blocking lock that modifies the status invariant and wakes up all parked waiters in a socket, and 2) CST-WS is also blocking but wakes up the selected number of parked waiters in which the number of wake-ups is equal to the number of hardware threads in a socket. At 120 threads, the native `mutex` suffers from cache-line bouncing and later from

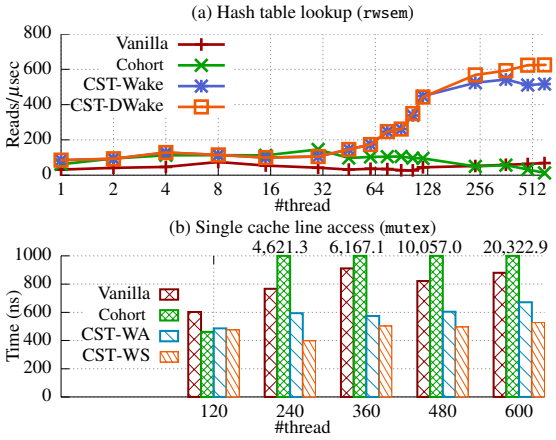


Figure 7: Two micro-benchmarks to illustrate the performance impact of various techniques employed by `CST-rwsem` and `CST-mutex`. Figure (a) represents the lookup performance of a concurrent hash table for 10% writes, which uses `rwsem`. Figure (b) shows the time taken to update a single cache line by holding a `mutex` with increasing thread count.

contention on its global `parking_list` while still maintaining a permissible performance beyond 120 threads. On the other hand, all CST variants address the cache-line bouncing issue for 120 threads. However, the Cohort lock suffers from spinning at higher core count since the waiters preempt the lock-holder after 120 threads. CST-WA and CST-WS address the limitation of the Cohort lock and maintain on-par performance even beyond 120 cores. CST-WS further mitigates the lock-holder preemption problem, since it does not wake up all waiters in one shot inside a socket, which has slightly higher throughput than CST-WA. In summary, CST-WS outperforms the Linux version by 1.7× at 6× over-subscription.

7.4 Critical Section Latency

We evaluate the lock/unlock pair latency of `rwsem` to gauge the effectiveness of CST against the Linux version. Table 1 shows that while NUMA-aware lock is a better fit for multiple readers/writers, it suffers in the low contention scenario because of the costly operation of finding the snode for readers and multiple atomic operations to obtain the lock, which we can improve with the hysteresis-based technique [5].

8 Discussion and Limitations

The current design of CST locks can introduce starvation in two cases: 1) re-queueing of the waiters after they are parked, and 2) the writer-preferred version of the `rwsem`. Although, in theory, we can devise a non-blocking algorithm that mitigates the overhead of costly scheduler interaction for the first case, we have not come across such an algorithm in practice and the CST lock is a better alternative than the current `mutex` that also suffers from the same starvation issue. We believe that this can be a plausible future research direction both in the terms of synchronization primitives and lightweight scheduling. For

Latency	RW-lock (ns)		
	Kernel	Cohort	CST
Reader (1 reader)	30.4	36.4	37.6
Reader (120 readers)	20,062.2	1,973.2	1,925.3
Writer (0 reader)	31.3	140.3	75.0
Writer (119 readers)	28,545.2	11,314.4	4,252.2

Table 1: Empty critical section latency for `rwsem`.

`CST-rwsem`, we choose a writer-preferred version because it batches readers, thereby improving the throughput of the application, which is similar to the design ideology of the Linux `rwsem` [37]. We can address this limitation by exactly adopting the writer-preferred version of the read-write Cohort lock [4].

Even though CST locks outperform both Cohort locks and the Linux `mutex`, we can further scale applications by using combining [13, 32] or the remote-core locking approach [23]. However, the only caveat with these approaches is that we need to rewrite some parts of the OS, which is not easy due to the large code base and complicated lock usage. Another area in which we can improve the performance of CST locks is the latency in low contention (Table 1). We are investigating the use of hardware transactional memory (TSX) to acquire and release the locks in a transaction as in prior work [5]. Although CST locks cannot completely replace all of the locks, they are beneficial to a few data structures that are critical and contend as much as `inode`, `mm`, `dentry`, etc.

9 Conclusion

Synchronization primitives are the basic building blocks of any parallel application, out of which the blocking synchronization primitives are designed to handle both over- and under-subscribed scenarios. We find that the existing primitives have sub-optimal performance for machines with large core count. They suffer either from cache-line contention or scheduler intervention in both scenarios, and are oblivious to the existing NUMA machines. In this work, we present scalable NUMA-aware, memory-efficient blocking primitives that exploit the NUMA hardware topology along with scheduling-aware parking and wake-up strategies. We implement `CST-mutex` and `CST-rwsem`, which provide the same benefit of existing non-blocking NUMA-aware locks in under-subscribed scenario while maintaining similar peak performance in over-subscribed cases. Our code is available here: https://github.com/sslab-gatech/cst-locks.

10 Acknowledgment

We thank the anonymous reviewers and our shepherd, Jean-Pierre Lozi, for their helpful feedback. This research was supported by the NSF award DGE-1500084, CNS-1563848, CRI-1629851, ONR under grant N000141512162, DARPA TC program under contract No. DARPA FA8650-15-C-7556, DARPA XD3 program under contract No. DARPA HR0011-16-C-0059, and ETRI MSIP/IITP[B0101-15-0644].

References

[1] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2010.

[2] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2012.

[3] D. Bueso and S. Norton. An Overview of Kernel Lock Improvements, 2014. https://events.linuxfoundation.org/sites/events/files/slides/linuxcon-2014-locking-final.pdf.

[4] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit. NUMA-aware Reader-writer Locks. In *Proceedings of the 18th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 157–166, Shenzhen, China, Feb. 2013.

[5] M. Chabbi and J. Mellor-Crummey. Contention-conscious, Locality-preserving Locks. In *Proceedings of the 21st ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 22:1–22:14, Barcelona, Spain, Mar. 2016.

[6] M. Chabbi, M. Fagan, and J. Mellor-Crummey. High Performance Locks for Multi-level NUMA Systems. In *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, San Francisco, CA, Feb. 2015.

[7] G. Chadha, S. Mahlke, and S. Narayanasamy. When Less is More (LIMO):Controlled Parallelism For improved Efficiency. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '12, 2012.

[8] D. Chinner. Re: [regression, 3.16-rc] rwsem: optimistic spinning causing performance degradation, 2014. https://lkml.org/lkml/2014/7/3/25.

[9] D. Dice. Malthusian Locks. *CoRR*, abs/1511.06035, 2015. URL http://arxiv.org/abs/1511.06035.

[10] D. Dice, V. J. Marathe, and N. Shavit. Flat-combining NUMA Locks. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 65–74, 2011.

[11] D. Dice, V. J. Marathe, and N. Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. In *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 247–256, New Orleans, LA, Feb. 2012.

[12] Facebook. A persistent key-value store for fast storage environments, 2012. http://rocksdb.org/.

[13] P. Fatourou and N. D. Kallimanis. Revisiting the Combining Synchronization Technique. In *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 257–266, New Orleans, LA, Feb. 2012.

[14] H. Guiroux, R. Lachaize, and V. Quéma. Multicore Locks: The Case is Not Closed Yet. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, pages 649–662, Denver, CO, June 2016.

[15] IBM. IBM K42 Group, 2016. http://researcher.watson.ibm.com/researcher/view_group.php?id=2078.

[16] *Xeon Processor E7-8890 v4 (60M Cache, 2.20 GHz)*. Intel, 2016. http://ark.intel.com/products/93790/Intel-Xeon-Processor-E7-8890-v4-60M-Cache-2_20-GHz.

[17] F. R. Johnson, R. Stoica, A. Ailamaki, and T. C. Mowry. Decoupling Contention Management from Scheduling. In *Proceedings of the 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 117–128, New York, NY, Mar. 2010.

[18] X. Leroy. The open group base specifications issue 7, 2016. http://pubs.opengroup.org/onlinepubs/9699919799/.

[19] R. Liu, H. Zhang, and H. Chen. Scalable Read-mostly Synchronization Using Passive Reader-writer Locks. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, pages 219–230, Philadelphia, PA, June 2014.

[20] Y. Liu. aim7 performance regression by commit 5a50508 report from LKP, 2014. https://lkml.org/lkml/2013/1/29/84.

[21] W. Long. qspinlock: Introducing a 4-byte queue spinlock, 2014. https://lwn.net/Articles/582897/.

[22] W. Long. locking/mutex: Enable optimistic spinning of lock waiter, 2016. https://lwn.net/Articles/696952/.

[23] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Fast and Portable Locking for Multicore Architectures. *ACM Trans. Comput. Syst.*, 33(4):13:1–13:62, Jan. 2016.

[24] V. Luchangco, D. Nussbaum, and N. Shavit. A Hierarchical CLH Queue Lock. In *Proceedings of the 12th International Conference on Parallel Processing*, Euro-Par'06, pages 801–810, 2006.

[25] 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.

[26] Microsoft. SQL Server 2014, 2014. http://www.microsoft.com/en-us/server-cloud/products/sql-server/features.aspx.

[27] 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)*, Denver, CO, June 2016.

[28] I. Molnar. Linux rwsem, 2006. http://www.makelinux.net/ldd3/chp-5-sect-3.

[29] I. Molnar and D. Bueso. Generic Mutex Subsystem, 2016. https://www.kernel.org/doc/Documentation/locking/mutex-design.txt.

[30] O. Nesterov. Linux percpu-rwsem, 2012. http://lxr.free-electrons.com/source/include/linux/percpu-rwsem.h.

[31] *Data Sheet: SPARC M7-16 Server*. Oracle, 2015. http://www.oracle.com/us/products/servers-storage/sparc-m7-16-ds-2687045.pdf.

[32] Y. Oyama, K. Taura, and A. Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA)*, pages 182–204, jul 1999.

[33] 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.

[34] SAP. SAP HANA 2: the transformer, 2015. http://hana.sap.com/abouthana.html.

[35] M. L. Scott. Non-blocking Timeout in Scalable Queue-based Spin Locks. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, pages 31–40, New York, NY, USA, 2002. ISBN 1-58113-485-1.

[36] M. L. Scott and W. N. Scherer. Scalable Queue-based Spin Locks with Timeout. In *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 44–52, Snowbird, Utah, June 2001.

[37] A. Shi. [PATCH] rwsem: steal writing sem for better performance,

2013. https://lkml.org/lkml/2013/2/5/309.

[38] L. Torvalds. Linux Wait Queues, 2005. http://www.tldp.org/LDP/tlk/kernel/kernel.html#wait-queue-struct.

[39] L. Torvalds. The Linux Kernel Archives, 2017. https://www.kernel.org/.

[40] A. Viro. parallel lookups, 2016. https://lwn.net/Articles/684089/.

[41] 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, 2010.

[42] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors. *ACM Comput. Surv.*, 45(1), Dec. 2012.

StreamBox: Modern Stream Processing on a Multicore Machine

Hongyu Miao[1], Heejin Park[1], Myeongjae Jeon[2],
Gennady Pekhimenko[2], Kathryn S. McKinley[3], and Felix Xiaozhu Lin[1]
[1]*Purdue ECE* [2]*Microsoft Research* [3]*Google*

Abstract

Stream analytics on real-time events has an insatiable demand for throughput and latency. Its performance on a single machine is central to meeting this demand, even in a distributed system. This paper presents a novel stream processing engine called StreamBox that exploits the parallelism and memory hierarchy of modern multicore hardware. StreamBox executes a pipeline of transforms over records that may arrive out-of-order. As records arrive, it groups the records into ordered epochs delineated by watermarks. A watermark guarantees no subsequent record's event timestamp will precede it.

Our contribution is to produce and manage abundant parallelism by generalizing out-of-order record processing within each epoch to out-of-order epoch processing and by dynamically prioritizing epochs to optimize latency. We introduce a data structure called *cascading containers*, which *dynamically* manages concurrency and dependences among epochs in the transform pipeline. StreamBox creates sequential memory layout of records in epochs and steers them to optimize NUMA locality. On a 56-core machine, StreamBox processes records up to 38 GB/sec (38M Records/sec) with 50 ms latency.

1 Introduction

Stream processing is a central paradigm of modern data analytics. Stream engines process unbounded numbers of records by pushing them through a pipeline of *transforms*, a continuous computation on records [3]. Records have *event* timestamps, but they may arrive out-of-order, because records may travel over diverse network paths and computations on records may execute at different rates. To communicate stream progression, transforms emit timestamps called *watermarks*. Upon receiving a watermark w_{ts}, a transform is guaranteed to have observed all prior records with *event* time $\leq ts$.

Most stream processing engines are distributed because they assume processing requirements outstrip the capabilities of a single machine [38, 28, 32]. However, modern hardware advances make a single multicore machine an attractive streaming platform. These advances include (i) high throughput I/O that significantly improves ingress rate, e.g., Remote Direct Memory Access (RDMA) and 10Gb Ethernet; (ii) terabyte DRAMs that hold massive in-memory stream processing state; and (iii) a large number of cores. This paper seeks to maximize streaming throughput and minimize latency on modern multicore hardware, thus reducing the number of required machines to process streaming workloads.

Stream processing on a multicore machine raises three major challenges. First, the streaming engine must extract parallelism aggressively. Given a set of transforms $\{d_1, d_2, \cdots, d_n\}$ in a pipeline, the streaming engine should exploit (i) pipeline parallelism by simultaneously processing all the transforms on different records in the data stream and (ii) data parallelism on all the available records in a transform. Second, the engine must minimize thread synchronization while respecting dependences. Third, the engine should exploit the memory hierarchy by creating sequential layout and minimizing data copying as records flow through various transforms in the pipeline.

To address these challenges, we present StreamBox, an out-of-order stream processing engine for multicore machines. StreamBox organizes out-of-order records into *epochs* determined by *arrival time* at pipeline ingress and delimited by periodic *event time* watermarks. It manages all epochs with a novel parallel data structure called *cascading containers*. Each container manages an epoch, including its records and end watermark. StreamBox dynamically creates and manages multiple inflight containers for each transform. StreamBox links upstream containers to their downstream consuming containers. StreamBox provides three core mechanisms:

(1) StreamBox satisfies dependences and transform

correctness by tracking producer/consumer epochs, records, and watermarks. It optimizes throughput and latency by creating abundant parallelism. It populates and processes multiple transforms and multiple in progress containers per transform. For instance, when watermark processing is a long latency event, StreamBox is not stalled, because as soon as any subsequent records arrive, it opens new containers and starts processing them.

(2) StreamBox elastically maps software parallelism to hardware. It binds a set of worker threads to cores. (i) Each thread independently retrieves a set of records (a *bundle*) from a container and performs the transform, producing new records that it deposits to a downstream container(s). (ii) To optimize latency, it prioritizes the processing of containers with timestamps required for the next stream output. As is standard in stream processing, outputs are scoped by *temporal windows* that are scoped by watermarks to one or more epochs.

(3) StreamBox judiciously places records in memory by mapping streaming access patterns to the memory architecture. To promote sequential memory access, it organizes pipeline state based on the output window size, placing records in the same windows contiguously. To maximize NUMA locality, it explicitly steers streams to flow within local NUMA nodes rather than across nodes.

We evaluate StreamBox on six benchmarks with a 12-core and 56-core machine. StreamBox scales well up to 56 cores, and achieves high throughput (millions of records per second) and low latency (tens of milliseconds) on out-of-order records. On the 56-core system, StreamBox reduces latency by a factor of 20 over Spark Streaming [38] and matches the throughput of results of Spark and Apache Beam [3] on medium-size clusters of 100 to 200 CPU cores for *grep* and *wordcount*.

The full source code of StreamBox is available at `http://xsel.rocks/p/streambox`.

2 Stream model and background

This section describes our out-of-order stream processing model and terminology, summarized in Table 1.

Streaming pipelines A stream processing engine receives one or more streams of *records* and performs a sequence of transforms $\mathscr{D} = \{d_1, d_2, \cdots, d_n\}$ on the records $\mathscr{R}$. Each record $r_{ts} \in \mathscr{R}$ has a timestamp ts for temporal processing. A record has an *event* timestamp defined by its occurrence (e.g., when a sensor samples a geolocation). Ingress of a record to the stream engine determines its *arrival* timestamp.

Out-of-order streaming Because data sources are diverse, records travel different paths, and transforms operate at different rates, records may arrive *out-of-order* at the stream processing engine or to individual transforms.

Table 1: Terminology

Term	Definition
Stream	An unbounded sequence of records
Transform	A computation that consumes and produces streams
Pipeline	A dataflow graph of transforms
Watermark	A special event timestamp for marking stream progression
Epoch	A set of records arriving between two watermarks
Bundle	A set of records in an epoch (processing unit of work)
Evaluator	A worker thread that processes bundles and watermarks
Container	Data structure that tracks watermarks, epochs, and bundles
Window	A temporal processing scope of records

To achieve low latency, the stream engine must *continuously* process records and thus cannot stall waiting for event and arrival time to align. We adopt the out-of-order processing (OOP) [27] paradigm based on *windows* to address this challenge.

Watermarks and stream epochs Ingress and transforms emit strictly monotonic event timestamps called *watermarks* w_{ts}, as exemplified in Figure 1(a). A watermark guarantees no subsequent records will have an event time earlier than *ts*. At ingress, watermarks delimit ordered consecutive *epochs* of records. An epoch may have records with event timestamps greater than the epoch's end watermark due to out-of-order arrival. The stream processing engine may process records one at a time or in *bundles*.

We rely on stream sources and transforms to create watermarks based on their knowledge of the stream data [2, 3]. We do not inject watermarks (as does prior work [7]) to force output and manage buffering.

Pipeline egress Transforms define event-time *windows* that dictate the granularity at which to output results. Because we rely on watermarks to define streaming progression, the *rate* of egress is bounded by the rate of watermarks, since a transform can only close a window after it receives a watermark. We define the *output delay* in a pipeline from the time it first receives the watermark w_{ts} that signals the completion of the current window to the moment when it delivers the window results to the user. This critical path is implicit in the watermark timestamps. It includes processing any remaining records in epochs that precede w_{ts} and processing w_{ts} itself.

Programming model We use the popular model from timely dataflow [30], Google dataflow [3], and others. To compose a pipeline, developers declare transforms and define dataflows among transforms. This is exemplified by the following code that defines a pipeline for Windowed Grep, one benchmark used in our evaluation (§9).

```
// 1. Declare transforms
Source<string> source(/*config info*/);
FixedWindowInto<string> fwi(seconds(1));
WindowedGrep<string>wingrep(/*regexp*/);
Sink<string> sink();

// 2. Create a pipeline
Pipeline* p = Pipeline::create();
```

```
p->apply(source); //set source

// 3. Connect transforms together
connect_transform(source, fwi);
connect_transform(fwi, wingrep);
connect_transform(wingrep, sink);

// 4. Evaluate the pipeline
Evaluator eval(/*config info*/);
eval.run(p);  // run the pipeline
```

Listing 1: Pseudo code for Windowed Grep pipeline

To implement a transform, developers must define the following functions, as shown in Figure 1(b): (i) `ProcessRecord(r)` consumes a record `r` and may emit derived records. (ii) `ProcessWm(w)` consumes a watermark `w`, flushes the transform's internal state, and may emit derived records and watermarks. `ProcessWm(w)` is always invoked only after `ProcessRecord(r)` consumes all records in the current epoch.

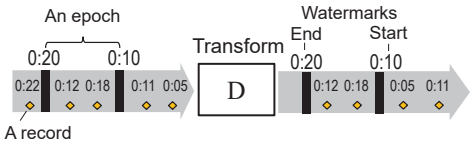
3 Design goals and criteria

We seek to exploit the potential of modern multicore hardware with its abundant hardware parallelism, memory capacity, and I/O bandwidth for high throughput and low latency. A key contribution of this paper is exploiting *epoch parallelism* by concurrently processing all available epochs in every transform, in addition to pipeline parallelism. Epoch parallelism generalizes the idea of processing the records in each epoch out-of-order by processing epochs out-of-order. The following two invariants ensure correctness:

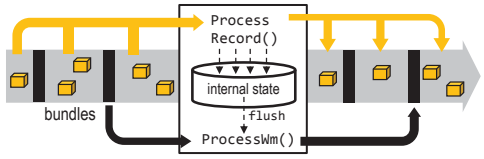
(1) *Records respect epoch boundaries* Each epoch is defined by a start watermark w_{start} and an end watermark w_{end} that arrive at ingress at time *start* and *end*, and consists only of records r_{at} that arrive at ingress at time *at*, with $start < at < end$. Once an ingress record r_{at} is assigned an epoch, records never changed epochs, since this change might violate the watermark guarantee.

(2) *Watermark ordering* A transform D may only consume w_{end} after it consumes all the records r in the epoch. This invariant transitively ensures that watermarks and epochs are processed in order, and is critical to pipeline correctness, as it enforces the progression contract on ingress and between transforms.

Our primary design goal is to minimize latency by exploiting epoch and pipeline parallelism with minimal synchronization while maintaining these invariants. In particular, our engine processes unconsumed records using all available hardware resources regardless of record ordering, delayed watermarks, or epoch ordering. We further minimize latency by exploiting the multicore memory hierarchy (i) by creating sequential memory layout and minimizing data movement, and (ii) by mapping streaming data flows to the NUMA architecture.



(a) A logic view of a transform, which consumes and produces out-of-order records as delimited by watermarks



(b) A transform in StreamBox. It is executed to consume and produce records belonging to multiple epochs in parallel.

Figure 1: A transform in a StreamBox pipeline.

4 StreamBox overview

A StreamBox pipeline includes multiple transforms and each transform has multiple containers. Each container is linked to a container in a downstream transform or egress. Containers form a network pipeline organization, as depicted in Figure 2. Records, derived records, and watermarks flow through the network by following the links. A window consists of one or more epochs. The window size determines the output aggregation and memory layout, but otherwise does not influence how StreamBox manages epochs.

This dataflow pipeline network is necessary to exploit parallelism because parallelism emerges dynamically as a result of variation in record arrival times and the variation in processing times of individual records and watermarks for different transforms. For instance, records, based on their content, may require variable amounts of processing. Furthermore, it is typically faster to process a record than a watermark. However, exposing this abundant record processing parallelism and achieving low latency require prioritizing containers on the critical path through the network. StreamBox prioritizes records in containers with timestamps preceding the pipeline's upcoming output watermark. Otherwise, the scheduler processes records from transforms with the most open containers. StreamBox thus dynamically adds parallelism to the bottleneck transforms of the network to optimize latency.

StreamBox implements three core components:

Elastic pipeline execution StreamBox dynamically allocates worker threads (*evaluators*) from a pool to transforms to maximize CPU utilization. StreamBox pins each evaluator to a CPU core to limit contention. During execution, StreamBox dispatches pending records and watermarks to evaluators. An evaluator executes transform code (i.e., `ProcessRecord()` or `ProcessWm()`) and

produces new records and watermarks that further drive the execution of downstream transforms.

When dispatching records, StreamBox packs them into variable sized *bundles* for processing to amortize dispatch overhead and improve throughput. Bundles differ from batches in many other streaming engines [38, 32, 7]. First, bundle size is completely orthogonal to the transform logic and its windowing scheme. StreamBox is thus at liberty to vary bundle size dynamically per transform, trading dispatch latency for overhead. Second, dynamically packing records in bundles does not delay evaluators and imposes little buffering delay. StreamBox only produces sizable bundles when downstream transforms back up the pipeline.

Cascading containers Each container belongs to a transform and tracks one epoch, its state (*open, processing,* or *consumed*), the relationship between the epoch's records and its end watermark, and the output epoch(s) in the downstream consuming transform(s). Each transform owns a set of containers for its current input epochs. With this container state, executers may concurrently consume and produce records in *all* epochs without breaking or relaxing watermarks.

Pipeline state management StreamBox places records belonging to the same temporal windows (one or more adjacent epochs) in contiguous memory chunks. It adapts a bundle's internal organization of records, catering to data properties, e.g., the number of values per key. StreamBox steers bundles so that they flow mostly within their own NUMA nodes rather than across nodes. To manage transform internal state, StreamBox instantiates a contiguous array of *slides* per transform, where each slide holds processing results for a given event-time range, e.g., a window. Evaluators operate on slide arrays based on window semantics, which are independent of the epoch tracking mechanism – cascading containers. The slide array realization incurs low synchronization costs under concurrent access.

5 Cascading containers

Cascading containers track epochs and orchestrate concurrent evaluators (i) to consume all of an epoch's records before processing its end watermark, (ii) to consume watermarks in stream order, and (iii) to emit records derived from an upstream epoch into the corresponding downstream epoch(s).

Figure 2 shows the cascading container design. Each transform owns a set of input stream containers, one for each potential epoch. When StreamBox creates a container *uc*, it creates one downstream container *dc* (or more) for its output in the downstream transform(s) and links to it, causing a cascade of container creation. It

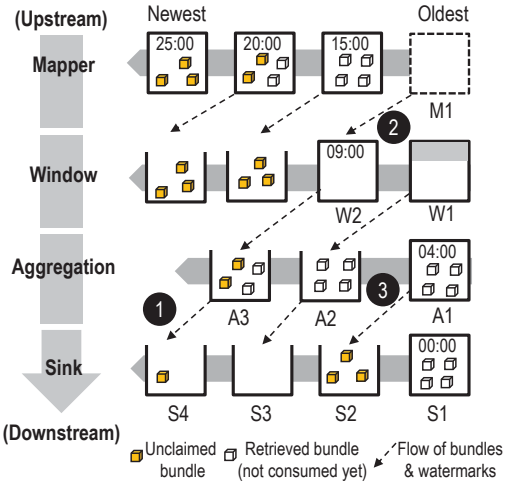


Figure 2: An overview of cascading containers

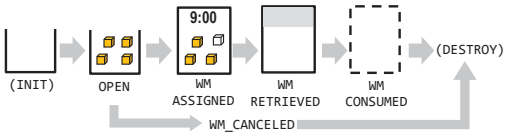


Figure 3: The life cycle of a container

puts all records and watermarks derived from the transform on *uc* into this corresponding downstream container *dc*. All these containers form a pipeline network. As stream processing progresses, the network topology evolves. Evaluators create new containers, establish links between containers, and destroy consumed containers.

5.1 Container implementation

StreamBox initializes a container D_{own} when the transform receives the first input record or bundle of an epoch. Each container includes any unclaimed bundles of the epoch. An *unconsumed* counter tracks the number of bundles that ever entered the container but are not fully consumed. After processing a bundle, D_{own} deposits derived output bundles in the downstream container and then updates the unconsumed counter.

Container state StreamBox uses a container to track an epoch's life cycle as follows and shown in Figure 3.

OPEN Containers are initially empty. An open container receives bundles from the immediate upstream D_{up}. The owner D_{own} processes the bundles simultaneously.

WM_ASSIGNED When D_{up} emits an epoch watermark w, it deposits w in D_{own}'s dependent container. Eventually D_{own} consumes all bundles in the container and the *unconsumed* counter drops to zero, at which point D_{own} retrieves and processes the end watermark.

WM_RETRIEVED A container enters this state when D_{own} starts processing the end watermark.

WM_CONSUMED After D_{own} consumes the end wat mark, it guarantees that it has flushed all derived st and the end watermark to the downstream contai and D_{own} may be destroyed.

WM_CANCELLED D_{up} chooses not to emit the end v termark for the (potential) epoch. Section 5.2 scribes how we support windowing transforms by c celling watermarks and merging containers.

Lock-free container processing Containers are lo free to minimize synchronization overhead. We instar ate the end watermark as an atomic variable that enfor acquire-release memory order. It ensures that D_{own} serves all D_{up} evaluators' writes to the container's claimed bundle set before observing D_{up}'s write of end watermark. The unclaimed bundle set is a concurr data structure that aggressively weakens the ordering mantics on bundles for scalability. Examples of ot such data structures include non-linearizable lock-f queues [13] and relaxed priority queues [4]. We further exploit this flexibility to make the bundle set NUMA-aware, as discussed in Section 7.1.

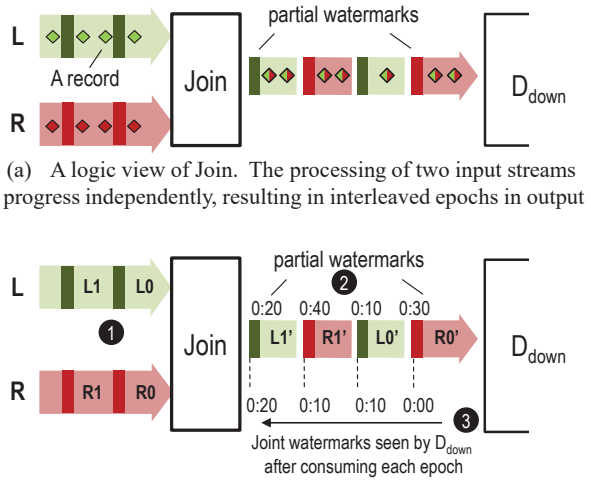
5.2 Single-input transforms

If a transform has only one input stream, all its input epochs – and therefore the containers – are ordered, even though records are not.

Creating containers The immediate upstream container D_{up} creates downstream containers on-demand and links to them. Figure 2 ❶ shows an example of container creation. When StreamBox processes the first bundle in A3, it creates S4 and any missing container that precedes it, in this case S3, and links A3 to S4 and A2 to S3. To make concurrent growth safe, StreamBox instantiates downstream links and containers using an atomic variable with strong consistency. Subsequent bundle processing uses the instantiated links and containers.

Processing To select a bundle to process, evaluators walk the container list for a transform, starting from the oldest container to the youngest, since the oldest container holds the most urgent work for state externalization. If an evaluator encounters containers in the `wm_consumed` state, it destroys the container. Otherwise,

1. it retrieves an unclaimed bundle. If none exists,
2. it retrieves the end watermark when (i) the watermark is valid (i.e., the container has `wm_assigned`), and (ii) all bundles are consumed (*unconsumed* == 0), and (iii) all watermarks in the preceding containers of D_{own} are consumed.
3. If the evaluator fails to retrieve a bundle or watermark from this container, it moves to the next younger container on D_{own}'s list.



(a) A logic view of Join. The processing of two input streams progress independently, resulting in interleaved epochs in output

(b) Arbitrarily ordering Join's output epochs relaxes watermarks

Figure 4: A logic diagram of OOP temporal join

Figure 2 shows an example. An evaluator starts from the oldest container W1 to find work ❷. Because W1 is in WM_RETRIEVED (all bundles are consumed and the end watermark is being processed), the worker moves on to W2. Because all bundles in W2 are consumed but the end watermark is available, it retrieves the watermark (09:00) for processing. Section 6 describes how we prioritize transforms in the container network.

Merging containers for windowing For each input container, we create a *potential* downstream container, expecting each input epoch will correspond to an output epoch. However, when a transform D performs windowing operations, it often must wait for multiple watermarks to correctly aggregate records. In this case, we merge containers. Figure 2 ❸ shows an example of Aggregation on a 10-min window. After consuming container A1 with its 04:00 watermark, the Aggregation transform cannot yet emit a watermark and retire its current window (0:00-10:00). Our solution is to cancel watermarks and merge the downstream output containers until the windowing logic, which uses event time, is satisfied. This operation is cheap. StreamBox cancels watermarks by simply marking them w_{cancel}. As evaluators walk container lists and observe w_{cancel}, they logically treat adjacent containers as one, e.g., S2 and S3. When the transform receives a watermark $ts \geq 10$, it emits the watermark which will eventually close the container.

5.3 Multi-input transforms

A multi-input transform, such as temporal Join and Union, takes multiple input streams and produces one stream. Figure 4 shows an example of out-of-order temporal join [27]. The left and right input streams progress independently (they share D_{join}'s internal state). The

output stream consists of *interleaved* epochs resulting from processing either input stream. These epochs are delimited by *partial* watermarks (w_L or w_R), which are also solely derived from the input streams. The downstream D_{down} derives a *joint* watermark as $min(w'_L, w'_R)$, where w'_L and w'_R are the most recent left and right partial watermarks.

The case for unordered containers A multi-input transform, unlike single-input transforms, cannot always have its downstream containers arranged on an *ordered* list (§5.2) because an optimal ordering of output epochs depends on their respective end (partial) watermarks. On the other hand, arbitrarily ordering output epochs may unnecessarily relax watermarks and delay watermark processing (§2).

Figure 4(b) shows an example of arbitrarily ordering output epochs. While processing *open* input epochs L0/L1 and R0/R1 ❶, StreamBox arbitrarily orders the corresponding output as L1'→R1'→L0'→R0' without knowing the end watermarks. Later, these output epochs eventually receive their *partial* end watermarks ❷. Upon consuming them, D_{down} derives joint watermarks based on its subsequent observations of partial watermarks ❸. Unfortunately, the joint watermark is more relaxed than the partial watermarks. For instance, the partial watermark 00:30 of R0' guarantees that all records in R0' are later than 00:30. However, from the derived joint watermark, D_{down} only knows that they are later than 00:00. Relaxed watermarks propagate to all downstream transforms. To tighten a joint watermark, StreamBox should have placed L0' and L1' (and perhaps more subsequent left epochs) before R0' and R1'. However, it cannot make that decision before observing all these partial watermarks!

In summary, StreamBox must achieve two objectives in tracking epochs for multi-input transforms. (1) It must track output epochs with corresponding containers for epoch parallelism. (2) It must defer ordering these containers until it determines their end watermarks.

Solution StreamBox maintains *unordered* containers for a multi-input transform's output epochs and their downstream counterparts. Once StreamBox determines the ordering of one epoch, it appends the corresponding container to an ordered list and propagates this change downstream. Figure 5 shows an example.

- D_{join} owns two ordered container lists L and R.
- D_1, the immediate downstream transform of D_{join}, owns three ordered lists of containers. L1 and R1 are derived from D_{join}'s L and R, respectively. S1 holds merged containers from L1 and R1.
- With D_2 downstream of D_1, D_2 owns an unordered set U and an ordered list S2.

As D_{join} processes its input streams L and R, it deposits the derived bundles and watermarks to containers on L1, R1, and S1 ❶. D_1 selects the oldest container C1 on L1 and R1 to process and it appends C1 to S1 ❷. Processing C1, deposits records in container C2 (following the *down* link), which subsequently produces records in containers at S2 ❸ and beyond ❹.

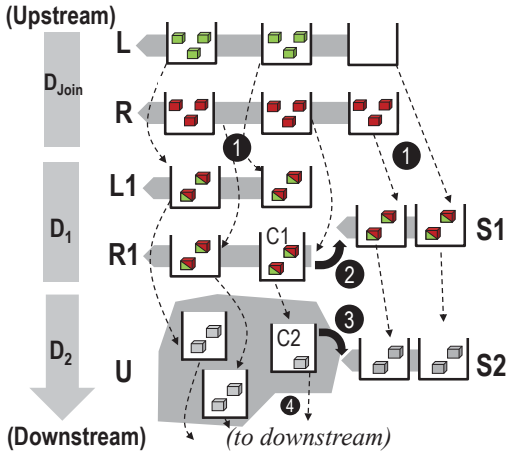


Figure 5: Unordered containers for Join and its downstream. For brevity, container watermarks are not drawn

5.4 Synchronized access to containers

In the cascading containers network, the concurrent evaluators dynamically modify the network topology by creating, linking, and destroying containers. Although the most frequent container operations, such as processing records, are lock-free as described in Section 5.1, modifying the container network must be synchronized. We carefully structure network modifications in reader and writer paths and synchronize them with one readers-writer lock for each container list. To retrieve work, an evaluator holds the container list's reader lock while walking the list. If the evaluator needs to modify the list (e.g., to destroy a container), it atomically upgrades the reader lock to a writer lock.

6 Pipeline scheduling

A pipeline's latency depends on how fast the engine externalizes the state of the current window. To this end, StreamBox's scheduler prioritizes upcoming state externalization.

StreamBox maintains a global notion of the *next externalization moment (NEM)*. The upcoming windowed output requires processing of all bundles and watermarks with timestamps prior to NEM. After each state externalization, StreamBox increments the NEM monotonically based on a prediction. In the common case where externalization is driven by temporal windows, the engine can

accurately predict NEM as the end of the current window. In case windowing information is unavailable, the engine may predict NEM based on historical externalization timing. Mispredicting NEM may increase the output delay but will not affect correctness.

NEM guides work prioritization in StreamBox. All evaluators independently retrieve work (i.e., bundles or watermarks) from cascading containers. By executing StreamBox's dispatch function, an evaluator looks for work by traversing container lists from the oldest to the youngest, starting from the top of the network. It prioritizes bundles in containers with timestamps that precede NEM.

Watermark processing is on the critical path of state externalization and often entails substantial amount of work, e.g., reduction of the window state. To accelerate watermark processing, StreamBox creates a special *watermark task queue*. Watermark tasks are defined as lambda functions. StreamBox gives these tasks higher priority and executes them with the same set of evaluators – without oversubscribing the CPU cores. An evaluator first processes watermark tasks. After completing a task, evaluators return to the dispatcher immediately. Evaluators never wait on a synchronization barrier inside the watermark evaluator. This on-demand, extra parallelism accelerates watermark evaluation.

7 Pipeline state management

The memory behavior of a stream pipeline is determined by the bundles of records flowing among transforms and the transforms' internal states. To manage this state, StreamBox targets locality, NUMA-awareness, and coarse-grained allocation/free. We decouple state management from other key aspects, including epoch tracking, worker scheduling, and transform logic.

7.1 Bundles

Adaptive internal structure StreamBox adaptively packs records into bundles for processing.

StreamBox seeks to (i) maximize sequential access, (ii) minimize data movement, and (iii) minimize the *per-record* overhead incurred by bundling.

A bundle stores a "flat" sequence of records sequentially in contiguous memory chunks. This logical record ordering supports grouping records temporally in epochs and windows, and by keys. It achieves both because temporal computation usually executes on all the keys of specific windows, rather than on specific keys of all windows. This choice contrasts to prior work that simply treats <window, key> as a new key.

To minimize data movement, StreamBox adapts bundle internals to the transform algorithm. For instance, given a Mapper that filters records, the bundles include both records and a bitmap, where each bit indicates the presence of a record, so that a record can be logically filtered by simply toggling a bit. Databases commonly use this optimization [7] as well.

StreamBox adapts bundle internals based on input data properties. The performance of keyed transforms, i.e., those consuming key-value pairs, is sensitive to the physical organization of these values. If each key has a large number of values, a bundle will hold a key's values using an array of pointers, each pointing to an array of values. This choice makes combining values produced by multiple workers as cheap as copying a few pointers. If each key only has a few values, StreamBox holds them in an array and copies them during combining. To learn about the input data, StreamBox samples a small fraction of it.

NUMA-aware bundle flows StreamBox explicitly steers bundles between transforms for NUMA locality by maximizing the chance that a bundle is both produced and consumed on the same NUMA node.

Each bundle resides in memory from one NUMA node and is labeled with that node. When an evaluator processes a container, it prefers unclaimed bundles labeled with its same NUMA node. It will process non-local bundles only when bundles from the local node are all consumed. To facilitate this process, an evaluator always allocates memory on its NUMA node, and later deposits the new bundle to the NUMA node of the downstream container. Notice that the NUMA-aware scheduling only affects the order among bundles within a container. It does not starve important work, e.g., containers to be dispatched by the next externalization moment.

7.2 Transform Internal State

StreamBox organizes a transform's internal state as an array of temporal *slides*, forming a slide. Each slide corresponds to a window (for fixed windows) or a window's offset (for sliding windows). Note that the size of a slide is independent of an epoch size.

To access a transform's state, an evaluator operates on a range of slides: updating slides in-place for accumulating processing results; fetching slides for closing a window; and retiring slides for state flushing. Since concurrent evaluators frequently access the slide arrays, we need to minimize locking and data movement. To achieve this goal, StreamBox grows the array on-demand and atomically. It only copies pointers when fetching slides. It decouples the logical retirement of slides from their actual, likely expensive destruction. To support concurrent access to a single slide, the current StreamBox implementation employs off-the-shelf concurrent data structures, as discussed below.

56CM	*Dell PowerEdge R930* 4x14 Xeon E7-4850v4 "Broadwell", 256GB DRAM, Linux 4.4
12CM	*Dell PowerEdge R720* 2x6 Xeon E5-2630v2 "Ivy Bridge", 256GB DRAM, Linux 4.4

Table 2: Test platforms used in experiments

8 Implementation

We implement StreamBox in 22K SLoC of C++11. The implementation extensively uses templates, static polymorphism, and C++ smart pointers. We implemented Windowing, GroupBy, Aggregation, Mapper, Reducer, and Temporal Join as our library transforms. Our scalable parallel runtime relies on the following scalable low-level building blocks.

C++ libraries We use boost [20] for timekeeping and locks, Intel TBB [22] for concurrent hash tables, and Facebook folly [17] for optimized vectors and strings. Folly improves the performance of some benchmarks by 20–30%. For scalable memory allocation, we use `jemalloc` [12], which scales much better than `std::alloc` and TBB [23] on our workloads.

Concurrent hash tables are hotspots in most statefull pipelines. We tested three open-source concurrent hash tables [22, 18, 17], but they either did not scale to a large core count or required pre-allocating a large amount of memory. Despite the extensive research on scalable hash tables [26, 6], we needed to implement an internally partitioned hash table. We wrapped TBB's concurrent hash map. This simple optimization improves our performance by 20–30%.

Bundle size is an empirical trade off between scheduling delay and overhead. StreamBox mainly varies bundle size at pipeline ingress. When the engine is fully busy, with all records in one ingress epoch, it produces as many bundles as evaluators, e.g., 56 bundles for 56 evaluators, to maximize the bundle size without starving any thread. The largest bundle size is around 80K records. When the ingress slows down, the system shrinks bundle sizes to reduce latency. We empirically determine that a $2\times$ reduction in bundle size balances a 10% drop in ingress data rate. We set the minimal bundle size at 1K records to avoid excessive per-record overhead.

9 Evaluation

Methodology We evaluate StreamBox on the two multicore servers, summarized in Table 2. 56CM is a high-end server that excels at real-time analytics and 12CM is a mid-range server. Although 100 Gb/s Infiniband (RDMA) networks are available, our local network is only 10 Gb/s. However, 10 Gb/s is insufficient to test StreamBox and furthermore even if we used Infiniband, it will directly store stream input in memory. We therefore generate ingress streams from memory. We dedicate a small number of cores (1–3) to the pipeline source. We then replay these large memory buffers pre-populated with records and emit in-memory stream epochs continuously. We measure the maximum sustained throughput of up to 38 GB/s at the pipeline source when the pipeline delay meets a given target.

Benchmarks We use the following benchmarks and datasets. Unless stated otherwise, each input epoch contains 1 M records and spans 1 second of event time. (1) **Windowed Grep (grep)** searches the input text and outputs all occurrences of a specific string. We use Amazon Movie Reviews (8.7 GB in total) [37] as input, a sliding window of 30 seconds, and 1 second target latency. The input record size is 1 KB. (2) **Word Count (wordcount)** splits input texts into words and counts the occurrences of each word. We use 954 MB English books [21] as input, a sliding window of 30 seconds, and 1 second target latency. The input record size is 100 bytes. (3) **Temporal Join (join)** has two input streams, for which we randomly generate unique 64-bit integers as keys. The join window for each record is $\pm$ 0.5 seconds. (4) **Counting Distinct URLs (distinct)** [32] counts unique URL identifiers. We use the `Yandex` dataset [16] with 70 M unique URLs and a fixed window of 1 second. (5) **Network Latency Monitoring (netmon)** [32] groups network latency records by IP pairs and computes the average per group. We use the Pingmesh dataset [19] with 88 M records and a fixed window of 1 second. The source emits 500K records per epoch. (6) **Tweets Sentiment Analysis (tweets)** [32] correlates sentiment changes in a tweet stream to the most frequent words. It correlates results from two pipelines: one that selects windows with significant sentiment score changes, and the other that calculates the most frequent words for each window. We use a public dataset of 8 million English tweets [10] and a fixed window of 1 second. This benchmark is the most complex and uses 8 transforms.

9.1 Throughput and Scalability

This section evaluates the throughput, scalability, and out-of-order handling of StreamBox, and compares with existing stream processing systems.

Throughput Figure 6 presents throughput on the y-axis for the six benchmarks as a function of hardware parallelism on the x-axis and latency as distinct lines. StreamBox has high throughput and typically processes millions of input records per second on a single machine, while delivering latencies as low as 50 ms. In particular, `grep` achieves up to 38 M records per second, which translates to 38 GB per second. This outstanding performance is due to low overheads and high system utilization. Profil-

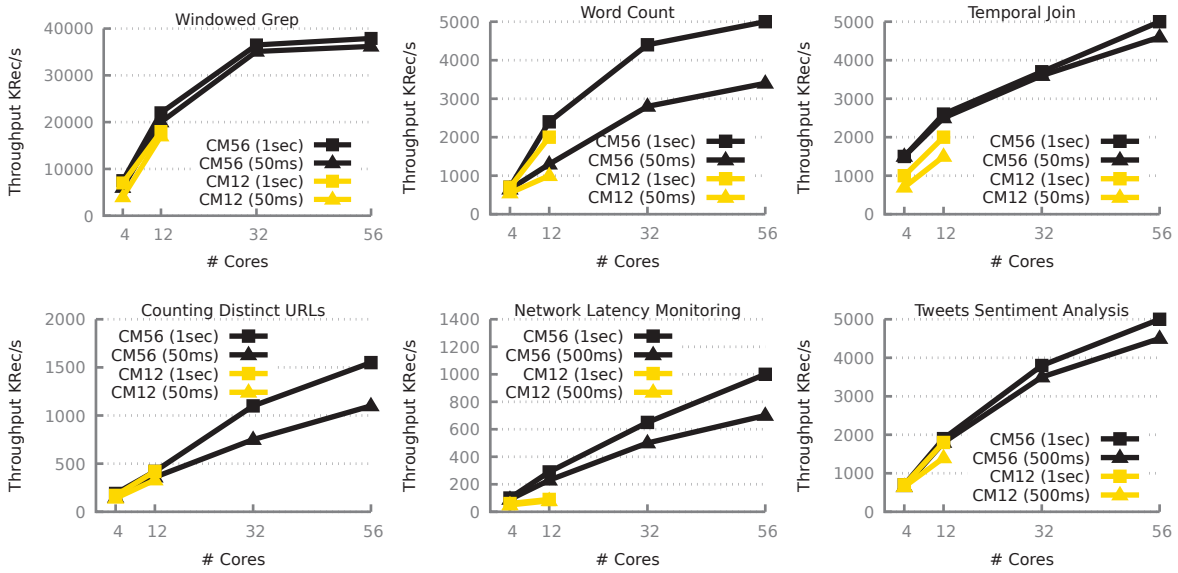
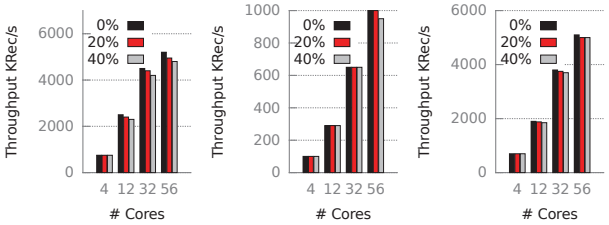


Figure 6: Throughput of StreamBox as a function of hardware parallelism and latency. StreamBox scales well.



(a) `wordcount` (b) `netmon` (c) `tweets`

Figure 7: StreamBox achieves high throughput even when a large fraction of records arrive out-of-order.

ing shows that all CPU cores have consistently high utilization ($>95\%$) and that most time is spent performing transform logic, e.g., processing stream data and manipulating hash tables.

Scalability Figure 6 shows that StreamBox scales well with core count for most benchmarks on both the 12-core and 56-core machines. When scalability diminishes in a few cases beyond 32 cores, as for `grep`, it is a result of memory-bound computation saturating the machine.

Out-of-order records By design, StreamBox efficiently computes on out-of-order records. To demonstrate this feature, we force a certain percent of records to arrive early in each epoch, i.e., the event time of these records is larger than the enclosing epoch's end watermark. Figure 7 shows the effect on throughput for 3 benchmarks. StreamBox achieves nearly the same throughput and latency as in in-order data processing. In particular, the throughput loss is as small as 7% even with 40% of records out-of-order. The minor degradation is due to early-arriving records that accumulate more windows in the pipeline. We attribute this consistent performance to (i) out-of-order epoch processing, since each transform continuously processes out-of-order records without delay, and (ii) prioritizing bundles and watermarks that decide the externalization latency of the current window in the scheduler.

Comparing to distributed stream engines We first compare StreamBox with published results of a few popular distributed stream processing systems and then evaluate two of them on our 56-core machine. Most published results are based on processing of in-order stream data. For out-of-order data, they either lack support (e.g., no notion of watermarks) or expect transforms to "hold and sort", which significantly degrades latency [11, 35].

Compared to existing systems, StreamBox jointly achieves low millisecond latency and high throughput (tens of millions of records per second). Very few systems achieve both. To achieve similar throughput, prior work uses at least a medium-size cluster with a few hundred CPU cores [28, 38]. For instance, under the 50-ms target latency, StreamBox's throughput on 56CM is 40× greater than StreamScope [28] running on 100 cores. Moreover, even under a 1-second target latency, StreamBox achieves much higher throughput per core. StreamBox can process 700K records/sec for `grep` and 90K records/sec for `wordcount` per core, which are 4.7× and 1.5× faster than the per-core processing rate reported by Spark Streaming on a 100-node cluster with a total of 400 cores.

We further experimentally compare StreamBox with Spark (v2.1.0) [38] and Apache Beam (v0.5.0) [3], on the same machine (56CM). We verify that they both utilize all cores. We set the the target latency to 1 second since they cannot achieve 50 ms as StreamBox does. Fig-

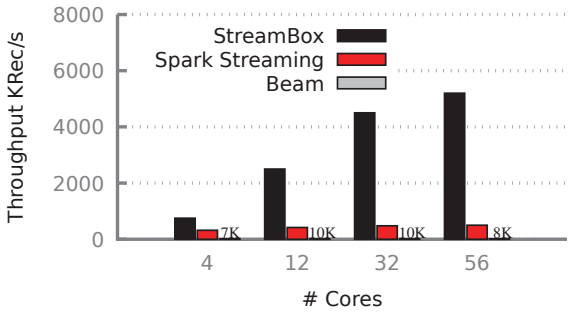
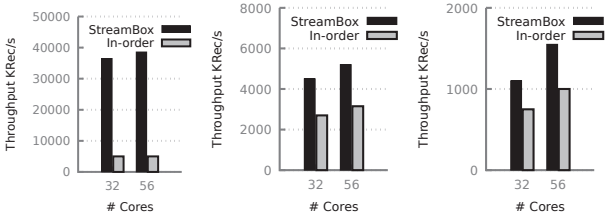


Figure 8: StreamBox scales better than Spark and Beam with `wordcount` on 56CM, with a 1-second target latency.



(a) `grep` (b) `wordcount` (c) `distinct`

Figure 9: In-order processing reduces parallelism, scalability, and throughput.

ure 8 shows that StreamBox achieves significantly higher throughput (by more than one order of magnitude) and it scales much better with core count.

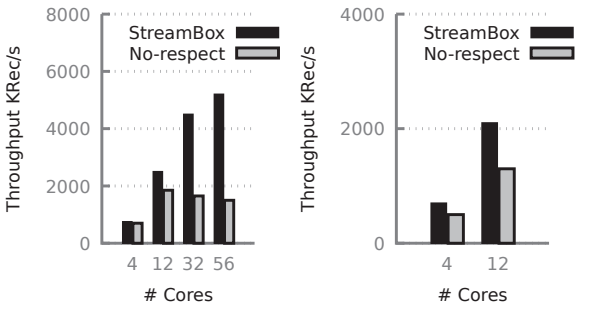
Comparing to single-machine streaming engines A few streaming engines are designed for a single machine: Oracle CEP [31], StreamBase [34], Esper [14], and SABER (for CPU+GPU) [24]. With 4 to 16 CPU cores, they achieve throughput between thousands and a few million of records per second. None of them reports to scale beyond 32 CPU cores. In particular, we tested Esper [14] on 56CM with `wordcount`. On four cores, Esper achieves around 900K records per second, which is similar to StreamBox with the same core count. However, we were unable to get Esper to scale even after applying recommended programming techniques, e.g., context partitioning [15]. As the core count increases, we observed the throughput drops.

In summary, StreamBox achieves better or similar per core performance than prior work. More importantly, StreamBox scales well to a large core count even with out-of-order record arrival.

9.2 Validation of key design features

This section evaluates the performance and scaling contributions of our key design features.

Epoch parallelism for out-of-order processing Epoch parallelism is fundamental to producing abundant parallelism and exploiting out-of-order processing. We compare with in-order epoch processing by implementing "hold and sort," in which each transform waits to process an epoch until all its records arrive. Note that this in-order epoch processing leaves out the high cost of sorting records. It processes records within an epoch out-of-order. Figure 9 shows that in-order epoch processing reduces throughput by 25% – 87%. Profiling reveals the reduced parallelism causes poor CPU utilization.



(a) `wordcount` on 56CM (b) `wordcount` on 12CM

Figure 10: When records do not respect epoch boundaries, it limits parallelism, scalability, and throughput.

Records must respect epoch boundaries (§3). StreamBox enforces the invariant that records respect epoch boundaries by mapping upstream containers to downstream containers (§5). We compare this to an alternative design where a transform's output records always flow into the *most recently* opened downstream container. Records then no longer respect epoch boundaries, since later records may enter earlier epochs. Violating the epoch invariant leads to huge latency fluctuations in watermark externalization, degrading performance. Figure 10 shows that not respecting epoch boundaries reduces throughput by up to 71%.

Prioritized scheduling (§6) Prioritizing containers on the critical path is crucial to latency and throughput. To explore its effect, we disable prioritized scheduling such that evaluators freely retrieve available bundles anywhere in the pipeline starting from its curent source and sink container. In this configuration, evaluators tend to rush into one transform, drain bundles there, and then move to the next. We confirmed this behavior with profiling. Performance measurements show that the pipeline latency fluctuates greatly and sometimes overshoots the target latency by a factor of 10.

NUMA-awareness (§7) We find NUMA-awareness especially benefits memory-bound benchmarks. For example, grep without windowing achieves 54 GB/s on 56CM, which is 12.5% higher than a configuration with NUMA-unaware evaluators.

Watermark arrival rates. Frequent watermarks lead to shorter epochs and more containers, each with fewer records, thus increasing the maintenance cost of cascad-

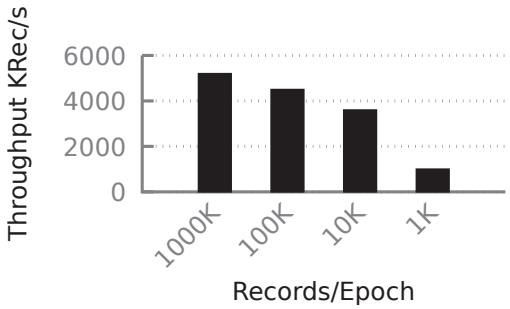


Figure 11: Performance impact of watermark arrival rate for `wordcount` on 56CM.

ing containers. In general, as shown in Figure 11, containers are sufficiently lightweight so that frequent watermarks (e.g., 100× more watermarks in 10K records/epoch) result in only a minor performance loss (e.g., 20%). However, substantial performance degradation emerges for watermarks at the rate of 1 K records/epoch, because frequent container creation and destruction incur too much synchronization.

10 Related work

This section compares StreamBox to prior work that uses the out-of-order processing (OOP) model, distributed and single server stream engines, and on exploiting shared memory for streaming.

OOP stream processing A variety of classic streaming engines focus on processing in-order records with a single core (e.g., StreamBase [34], Aurora [1], TelegraphCQ [9], Esper [14], Gigascope [11], and NiagaraST [29]). Li et al. [27] advocate OOP stream processing that relies on stream progression messages, e.g. punctuations, for better throughput and efficiency. The notion of punctuations is implemented in many modern streaming engines [3, 7, 28]. These systems do exploit pipeline and batch parallelism, but they do not exploit out-of-order processing of epochs to expose and deliver highly scalable data parallelism on a single server.

Single-machine streaming engines Trill [8] inspires StreamBox's state management with its columnar store and bit-vector design. However, Trill's punctuations are generated by the engine itself in order to flush its internal batches, which limits parallelism. Furthermore, Trill assumes ordered input records, which limits it applicability. StreamBox has neither of these limitations. SABER [24] is a hybrid streaming engine for CPUs and GPGPUs. Similar to StreamBox, it exploits data parallelism with multithreading. However, SABER does not support OOP. It must reorder execution results from concurrent workers, limiting its applicability and scalability. Oracle CEP [31] exploits record parallelism by relaxing record ordering. However, it lacks the notion of watermarks and does not implement statefull OOP pipelines.

Distributed streaming engines Several systems process large continuous streams using hundreds to thousands of machines. Their designs often focus on addressing the pressing concerns of a distributed environment, such as fault tolerance [38, 32, 28], programming models [30, 3], and API compatibility [36]. TimeStream [32] tracks data dependence between transform's input and output, but uses it for failure recovery. StreamBox also tracks fine-grained epoch dependences, but for minimizing externalization latency. StreamScope [28] handles OOP using watermark semantics, but it does not exploit OOP for performance as does StreamBox. It instead implements operator determinism based on holding and waiting for watermarks. StreamBox is partially inspired by Google's dataflow model [3] and is an implementation of its OOP programming model. However, to the best of our knowledge and based on our experiments, the Apache Beam [5] open-source implementation of Google dataflow does not exploit epoch parallelism on a multicore machine.

Data analytics on a shared memory machine Some data analytics engines propose to facilitate sequential memory access [33, 25] and one exploits NUMA [39]. StreamBox's bundles are similar to morsels in a relational query evaluator design [26], where evaluators process data fragments ("morsels") in batch and that are likely allocated on local NUMA nodes. StreamBox favors low scheduling delay for stream processing. Evaluators are rescheduled after consuming each bundle, instead of executing the entire pipeline for that bundle.

11 Conclusions

This paper presents the design of a stream processing engine that harnesses the hardware parallelism and memory hierarchy of modern multicore servers. We introduce a novel data structure called *cascading containers* to track dependences between epochs while at the same time processing any available records in any epoch. Experimental results show StreamBox scales to a large number of cores and achieves throughput on-par with distributed engines on medium-size clusters. At the same time, StreamBox delivers latencies in the tens of milliseconds, which are 20× shorter than other large-scale streaming engines. The key contribution of our work is a generalization of out-of-order record processing to out-of-order epoch processing that maximizes parallelism while minimizing synchronization overheads.

Acknowledgments

This work was supported in part by NSF Award #1619075 and by a Google Faculty Award. The authors thank the anonymous reviewers and the paper shepherd, Charlie Curtsinger, for their useful feedback.

References

[1] ABADI, D., CARNEY, D., CETINTEMEL, U., CHERNIACK, M., CONVEY, C., ERWIN, C., GALVEZ, E., HATOUN, M., MASKEY, A., RASIN, A., ET AL. Aurora: a data stream management system. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (2003), ACM, pp. 666–666.

[2] AKIDAU, T., BALIKOV, A., BEKIROĞLU, K., CHERNYAK, S., HABERMAN, J., LAX, R., MCVEETY, S., MILLS, D., NORDSTROM, P., AND WHITTLE, S. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow. 6*, 11 (Aug. 2013), 1033–1044.

[3] 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.

[4] ALISTARH, D., KOPINSKY, J., LI, J., AND SHAVIT, N. The spraylist: A scalable relaxed priority queue. *SIGPLAN Not. 50*, 8 (Jan. 2015), 11–20.

[5] APACHE. Beam. `https://beam.apache.org/`, 2017.

[6] BALKESEN, C., TEUBNER, J., ALONSO, G., AND ÖZSU, M. T. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on* (2013), IEEE, pp. 362–373.

[7] CHANDRAMOULI, B., GOLDSTEIN, J., BARNETT, M., DELINE, R., FISHER, D., PLATT, J. C., TERWILLIGER, J. F., AND WERNSING, J. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment 8*, 4 (2014), 401–412.

[8] CHANDRAMOULI, B., GOLDSTEIN, J., BARNETT, M., DELINE, R., FISHER, D., PLATT, J. C., TERWILLIGER, J. F., AND WERNSING, J. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment 8*, 4 (2014), 401–412.

[9] 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.

[10] CHENG, Z., CAVERLEE, J., AND LEE, K. You are where you tweet: a content-based approach to geo-locating twitter users. In *Proceedings of the 19th ACM international conference on Information and knowledge management* (2010), ACM, pp. 759–768.

[11] CRANOR, C., JOHNSON, T., SPATASCHEK, O., AND SHKAPENYUK, V. Gigascope: a stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (2003), ACM, pp. 647–651.

[12] DAVID GOLDBLATT, DAVE WATSON, J. E. Jemalloc memory allocator. `http://http://jemalloc.net/`, 2017.

[13] DESROCHERS, C. moodycamel::concurrentqueue. `https://github.com/cameron314/concurrentqueue`, 2016.

[14] ESPERTECH. Esper. `http://www.espertech.com/esper/`, 2017.

[15] ESPERTECH. Esper faq. `http://www.espertech.com/esper/faq_esper.php#scaling`, 2017.

[16] EUGENE KHARITONOV, P. S. Yandex: Personalized web search challenge. `https://www.kaggle.com/c/yandex-personalized-web-search-challenge/data`, 2017.

[17] FACEBOOK. Folly. `https://github.com/facebook/folly#folly-facebook-open-source-library`, 2017.

[18] GOYAL, M., FAN, B., LI, X., ANDERSEN, D. G., AND KAMINSKY, M. Libcuckoo. `https://github.com/efficient/libcuckoo`, 2017.

[19] GUO, C., YUAN, L., XIANG, D., DANG, Y., HUANG, R., MALTZ, D., LIU, Z., WANG, V., PANG, B., CHEN, H., ET AL. Pingmesh: A large-scale system for data center network latency measurement and analysis. *ACM SIGCOMM Computer Communication Review 45*, 4 (2015), 139–152.

[20] GURTOVOYI, A., AND ABRAHAMSI, D. Boost c++ libraries. http://www.boost.org/, 2017.

[21] HART, M. Free ebooks by project gutenberg. http://www.gutenberg.org/wiki/Main_Page, 2017.

[22] INTEL. Intel threading building blocks. https://software.intel.com/en-us/intel-tbb, 2017.

[23] INTEL. Scalable memory allocator. https://www.threadingbuildingblocks.org/tutorial-intel-tbb-scalable-memory-allocator, 2017.

[24] KOLIOUSIS, A., WEIDLICH, M., CASTRO FERNANDEZ, R., WOLF, A. L., COSTA, P., AND PIETZUCH, P. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD '16, ACM, pp. 555–569.

[25] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. Graphchi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 31–46.

[26] LEIS, V., BONCZ, P., KEMPER, A., AND NEUMANN, T. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data* (2014), ACM, pp. 743–754.

[27] LI, J., TUFTE, K., SHKAPENYUK, V., PAPADIMOS, V., JOHNSON, T., AND MAIER, D. Out-of-order processing: a new architecture for high-performance stream systems. *Proceedings of the VLDB Endowment 1*, 1 (2008), 274–288.

[28] LIN, W., QIAN, Z., XU, J., YANG, S., ZHOU, J., AND ZHOU, L. Streamscope: continuous reliable distributed processing of big data streams. In *Proc. of NSDI* (2016), pp. 439–454.

[29] MAIER, D., LI, J., TUCKER, P., TUFTE, K., AND PAPADIMOS, V. Semantics of data streams and operators. In *International Conference on Database Theory* (2005), Springer, pp. 37–52.

[30] 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* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 439–455.

[31] ORACLE. Stream explorer. http://bit.ly/1L6tKz3, 2017.

[32] 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 *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 1–14.

[33] 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* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 472–488.

[34] STANLEY ZDONIK, MICHAEL STONEBRAKER, M. C. Streambase systems. http://www.tibco.com/products/tibco-streambase, 2017.

[35] TUCKER, P. A., MAIER, D., SHEARD, T., AND FEGARAS, L. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering 15*, 3 (2003), 555–568.

[36] TWITTER. Heron. https://twitter.github.io/heron/, 2017.

[37] WANG, L., ZHAN, J., LUO, C., ZHU, Y., YANG, Q., HE, Y., GAO, W., JIA, Z., SHI, Y., ZHANG, S., ET AL. Bigdatabench: A big data benchmark suite from internet services. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on* (2014), IEEE, pp. 488–499.

[38] ZAHARIA, M., DAS, T., LI, H., HUNTER, T., SHENKER, S., AND STOICA, I. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 423–438.

[39] ZHANG, K., CHEN, R., AND CHEN, H. Numa-aware graph-structured analytics. *SIGPLAN Not. 50*, 8 (Jan. 2015), 183–193.

Everything you always wanted to know about multicore graph processing but were afraid to ask

Jasmina Malicevic
EPFL

Baptiste Lepers
EPFL

Willy Zwaenepoel
EPFL

Abstract

Graph processing systems are used in a wide variety of fields, ranging from biology to social networks, and a large number of such systems have been described in the recent literature. We perform a systematic comparison of various techniques proposed to speed up in-memory multicore graph processing. In addition, we take an end-to-end view of execution time, including not only algorithm execution time, but also pre-processing time and the time to load the graph input data from storage.

More specifically, we study various data structures to represent the graph in memory, various approaches to pre-processing and various ways to structure the graph computation. We also investigate approaches to improve cache locality, synchronization, and NUMA-awareness. In doing so, we take our inspiration from a number of graph processing systems, and implement the techniques they propose in a single system. We then selectively enable different techniques, allowing us to assess their benefits in isolation and independent of unrelated implementation considerations.

Our main observation is that the cost of pre-processing in many circumstances dominates the cost of algorithm execution, calling into question the benefits of proposed algorithmic optimizations that rely on extensive pre-processing. Equally surprising, using radix sort turns out to be the most efficient way of pre-processing the graph input data into adjacency lists, when the graph input data is already in memory or is loaded from fast storage. Furthermore, we adapt a technique developed for out-of-core graph processing, and show that it significantly improves cache locality. Finally, we demonstrate that NUMA-awareness and its attendant pre-processing costs are beneficial only on large machines and for certain algorithms.

1 Introduction

Interest in processing graph-structured data has grown over the last few years, especially for mining relationships in social network graphs. Many graph processing systems have been built, including single-machine, cluster-based, in-memory and out-of-core systems [7, 8, 12–14, 16, 17, 19, 20, 22, 23, 26, 27, 29, 33, 36, 37]. In this paper we focus on single-machine in-memory graph processing systems. With the recent increase in main memory size and number of cores, such machines can now process very large graphs in a reasonable amount of time.

With few exceptions [4, 28], most papers on graph processing systems present a new system and compare its performance (and occasionally its programmability) to previous systems. While interesting, these comparisons are often difficult to interpret, because systems are multi-dimensional, and therefore a variety of features may contribute to observed performance differences. Variations in hardware and software infrastructure, input formats, algorithms, graphs and measurement methods further obscure the comparison.

In this paper we take a different approach. Rather than comparing different systems, we compare different techniques used in graph processing systems, and we try to answer the question: what techniques provide what benefits for what types of algorithms and graphs? We implement various techniques proposed in different papers in a single system. We then selectively enable the different techniques, and compare the performance of the resulting approach on the same hardware platform for the same algorithms and graphs.

In particular we take an end-to-end view of graph processing, often absent in other papers. Graph processing involves loading the graph as an edge array from storage, pre-processing the input to construct the necessary data structures, executing the actual graph algorithm, and storing the results. Most papers focus solely on the algo-

rithm phase, but we demonstrate that there is an important trade-off between pre-processing time and algorithm execution time. While we recognize that pre-processing can potentially be amortized over repeated executions, we show that gains in algorithm execution time can be completely undone by increases in pre-processing time.

We structure our investigation of algorithm execution time along two dimensions. In a first dimension, we distinguish between a vertex-centric approach, in which the algorithm iterates over vertices, and an edge-centric approach, in which the algorithm iterates over edges. In addition, we propose a new iteration approach, adapted from out-of-core systems [37], in which the algorithm iterates over grids, with improved cache locality as a result. In a second dimension, we distinguish between algorithms that push information to their neighbors, or pull information from them. We also consider algorithms that dynamically choose between push and pull.

To illustrate through a simple example the importance of an end-to-end view, we analyze the push-pull approach to Breadth First Search (BFS)[1]. Earlier papers [2, 3, 29] have demonstrated that, for BFS, a push-pull approach results in better algorithm execution time than the conventional push approach. Figure 1 shows the end-to-end execution time of BFS on the well-known Twitter follower graph [18] using both approaches. While the algorithm execution time is indeed 3× smaller for push-pull, the overall execution is completely dominated by pre-processing. The pre-processing time is 2× larger for push-pull, resulting in 1.5× worse overall end-to-end time.

In addition to different methods of iteration and information flow, various optimizations have been proposed to take advantage of memory locality on NUMA machines. These optimizations often take the form of partitioning data structures during pre-processing, such that most accesses during algorithm execution are local to a NUMA node. Continuing the theme of the trade-off of pre-processing versus algorithm execution times, we investigate whether such pre-processing pays off for graph processing.

The main results of this paper are:

- An illustration of the fundamental trade-off between pre-processing and algorithm execution time in graph processing.
- An evaluation of different techniques for building adjacency lists, showing that radix sort provides the best performance when the graph is in memory or when it is loaded from a fast storage medium.
- An evaluation of the pre-processing vs. algorithm execution time trade-off for vertex-centric vs. edge-centric computation, showing that the construction of adjacency lists for vertex-centric processing may or may not pay off, depending on the algorithm execution time.
- An evaluation of a push vs. pull information flow, illustrating the benefits of reduced computation for push vs. reduced synchronization for pull.
- An evaluation of the pre-processing vs. computation trade-off for combined push-pull information flow, showing that the extra pre-processing costs associated with this combination outweigh gains in algorithm execution time.
- The adaptation of an out-of-core technique for improving the cache locality and the synchronization overhead of an in-memory graph processing system.
- An evaluation of the pre-processing vs. computation tradeoff for NUMA-aware optimizations, demonstrating that their large pre-processing times can be compensated by gains in algorithm execution time only on large NUMA machines and only for certain algorithms.

[1] see Section 6 for a precise definition of push-pull

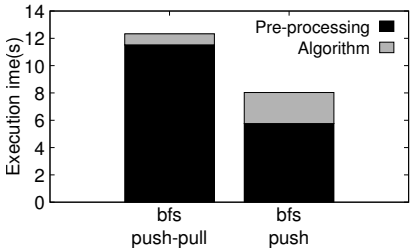


Figure 1: Example of the trade-off between pre-processing and algorithm execution time for BFS on the Twitter graph: push-pull improves algorithm execution time, but the required pre-processing time leads to overall worse end-to-end execution time (measured on Ligra [29]).

The outline of this paper is somewhat unusual. We start in Section 2 with an overview of the hardware and software used in this paper. We discuss data structures and pre-processing costs in Section 3. In Section 4 we look at the relationship between the data layout and vertex-centric or edge-centric computation. Section 5 discusses methods for improving cache locality. In Section 6 we evaluate the choice between push and pull approaches and its implications for algorithm execution time, pre-processing time and synchronization overhead. Section 7 evaluates graph partitioning approaches to take advantage of NUMA characteristics. Section 8 summarizes results on graphs and algorithms not discussed in previous sections. Section 9 provides an overview of all the results in one place. Section 10 discusses the graph processing systems from which we draw inspiration for this work. Section 11 concludes the paper.

The code used for the experiments in this paper and instructions on how to run them is available at: `https://github.com/epfl-labos/EverythingGraph`.

2 Experimental setup

Experimental environment. We evaluate the pre-processing and algorithm execution times on two machines, each representative of a large class of machines. Machine A has 2 NUMA nodes, and is less sensitive to NUMA effects than machine B, which has 4 NUMA nodes. More precisely, machine A has 2 Intel Xeon E5-2630 processors, each with 8 cores (16 cores in total) and a 20MB LLC cache, and 128GB of RAM. Machine B has 4 AMD Opteron 6272 processors, each with 8 cores (32 cores in total) and a 16MB LLC cache, and 256GB of RAM. Unless otherwise stated, all experiments are run on Machine B.

The pre-processing times, unless otherwise stated, assume the graph is already loaded in memory. The costs of loading the graph into memory and its implications on pre-processing are discussed separately.

The subset of vertices or edges to be processed during a computation step is kept in a work queue. Threads take work items from the queue in large enough chunks to reduce the work distribution overheads. We parallelize both pre-processing and computation using the Cilk 4.8 parallel runtime system. When needed, Cilk balances the work among threads by allowing threads to steal work items from one another. Our experiments using OpenMP and PThreads show comparable execution times and are therefore not reported.

Algorithms. We select six algorithms with different characteristics in terms of functionality (traversal, machine learning, ranking), vertex metadata, as well as the number of vertices active during computation steps (iterations).

We evaluate the following three traversal algorithms. **Breadth-first search (BFS)** traverses a graph from a given source vertex and builds a tree in breadth-first order. **Weakly connected components (WCC)** discovers connected vertices within a graph and classifies them into components using label propagation. **Single source shortest path (SSSP)** finds the (length of the) shortest path between a given source vertex and every other reachable vertex in the graph. We also evaluate two algorithms that compute over the entire graph: **Pagerank (PR)** [24] is a ranking algorithm used to rank web pages based on their popularity. **Sparse matrix vector multiplication (SpMV)** multiplies the adjacency matrix of a graph with a vector of values. The matrix entries are stored as edge weights. Finally, **Alternating Least Squares** (**ALS**) is an optimization method used in recommender systems.

Datasets. Table 1 gives an overview of the graphs used along with their number of vertices and edges. We use both synthetic and real-world datasets. The synthetic datasets are power-law graphs generated by the RMAT graph generator [5]. We generate graphs of different sizes to evaluate the scalability of optimizations in terms of graph size. RMAT26 is the biggest RMAT graph that we can fit on all machines for all approaches. As a real-world power-law dataset, we use the Twitter follower graph [18], which is the largest real-world dataset that fits on all machines for all approaches.

In addition to these two graphs, we also use the US-Road graph from the DIMACS challenge [1]. This graph has a different shape than power-law graphs: it has a high diameter, and all vertices have a small in/out degree. We use it to study the impact of the shape of the graph on different computation approaches. Finally, for ALS we use the bipartite Netflix graph [35].

Graph	Vertices	Edges
RMAT-N	2^N	2^{N+4}
Twitter	62M	1468M
US-Road	23.9M	58M
Netflix	0.5M	100M

Table 1: Graphs used in the evaluation, with their number of vertices and edges.

Due to space constraints, in Sections 3 to 7, we primarily present results for BFS and Pagerank (with 10 iterations). These algorithms represent opposite ends of the spectrum, both in terms of the percentage of the graph that is active during each step of the computation and in terms of computation complexity. Furthermore, we report results primarily for the RMAT26 graph. We include results for other algorithms and graphs only when they provide additional insights that depend on the algorithm or the shape of the graph. Section 8 completes the picture by presenting data on the combinations of algorithms and input graphs not discussed in earlier sections.

3 Data layouts and pre-processing costs

In this section we first present different data layouts and their associated pre-processing costs.

3.1 Data layouts

Edge arrays are the simplest and the default way to distribute graphs [27] and are used by many systems [6, 12, 27]. Graphs are stored as an array containing pairs of integers corresponding to the source and the destination vertex of each edge. In the remainder of the paper, we assume the graph input takes the form of an edge array and needs to be further converted into other formats.

Adjacency lists store edges in per-vertex edge arrays. Each vertex points to an array containing the destination vertices of its outgoing edges, and possibly also to an array containing the source vertices of its incoming edges.

3.2 Pre-processing costs

Edge array. The layout of edge arrays matches the format of the input file, and it suffices to map the input file in memory to be able to start computation. As such, edge arrays incur no pre-processing cost.

Adjacency lists. We explore two techniques to build adjacency lists.

The simplest technique consists of reading the input file and *dynamically* allocating and resizing the edge arrays of vertices as new edges are discovered.

The second technique avoids reallocations by loading the graph as an edge array and then sorting it by source vertex. Vertices use an index in the sorted edge array to point to their outgoing edge array. The incoming edge array is created by sorting the edge array by destination vertex. This way the edges are stored contiguously in memory, corresponding to compressed sparse row format (CSR). The performance of this approach depends on the sorting algorithm.

The most common approach to sort edges is to use a *count sort*. In a first pass over the edge array, we count the number of outgoing (incoming) edges for each vertex. In a second pass over the edge array, we place edges at the correct location in the sorted edge array. Most existing graph analytics frameworks use this approach, as it is optimal in terms of complexity (the input array is only scanned twice).

An alternative approach is based on *radix sort*. Radix sort treats keys as multi-digit numbers, and sorts the keys into buckets one digit at a time. In the parallel version, each thread recursively sorts a subset of edges into a small number of buckets [32]. The advantage of radix sort is that buckets are written sequentially, and therefore have good locality. The complexity of the sort is relatively low. We use a radix size of 8 bits (256 buckets) which only requires $log_2(\#vertices)/8$ recursions to sort the edge array (e.g., 4 recursions for a graph with 4 billion vertices, 8 recursions with 2^{64} vertices).

3.3 Evaluation

Table 2 presents, for all three approaches (dynamic, count sort and radix sort), the execution times for creating outgoing per-vertex edge arrays and for creating both incoming and outgoing per-vertex edge arrays, for the Twitter graph and assuming the graph is already in memory. Using a radix sort is 4.8× faster than count sort. Surprisingly, sorting using a radix sort is also 4.9× faster than dynamically building the per-vertex edge arrays. Radix sort is faster, because it has better cache locality than the other solutions. Both the dynamic approach and count sort sequentially read the input edge array, but the subsequent steps have poor cache locality. The dynamic approach requires jumping between per-vertex arrays to insert a newly read edge. Count sort requires jumping between vertices as well in order to count their degree. It then does another scan of the input to place edges at their corresponding offsets in the sorted edge array. This step jumps between distant positions in the array.

Adj. list pre-processing variation	Twitter out	Twitter in-out	LLC misses
Dynamic	20.0	27.2	69%
Count sort	19.5	23.9	71%
Radix sort	4.0	8.5	26%

Table 2: Adjacency list creation cost (in seconds) and percentage of LLC misses on machine B when the graph is in memory.

Figure 2 presents the evolution of the pre-processing time for RMAT graphs depending on the graph size. All approaches scale as the graph size increases. The radix sort approach is always faster than the count sort and the dynamic sort approach (3.3× and 3.8×, respectively, on RMAT26).

For smaller graphs, count sort is slower than both the dynamic and radix approaches. The approach requires reading the edge array twice (once for counting, and then once to place edges in the sorted array). As the graph grows, however, the fact that the second pass in count sort does no reallocations makes it slightly better than the dynamic approach (e.g. there are 32 million reallocations for an RMAT26 graph).

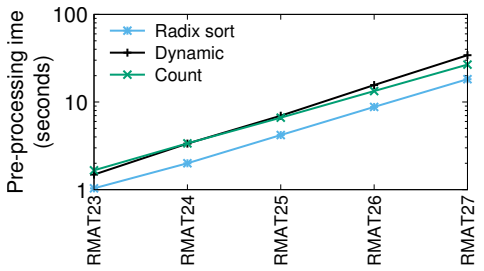


Figure 2: Scaling of pre-processing methods for adjacency list creation. All methods scale linearly with the graph size. RMAT-(N+1) is double the size of RMAT-N, and so is the pre-processing time.

3.4 Loading and pre-processing

The previous discussion assumes that the graph is already loaded into memory. Conclusions are different when the graph is to be read from storage or over the network. Indeed, doing a radix sort can only be partially overlapped with loading the graph in memory. In contrast, the dynamic approach of allocating and resizing

per-vertex edge arrays can be fully overlapped with loading. For count sort, only the first pass can be overlapped with loading.

3.5 Evaluation with loading included

Table 3 presents the combined loading and pre-processing time when the graph is loaded from an SSD (380MB/s maximum bandwidth) and from a regular hard drive disk (100MB/s).

If we take loading speed into account, dynamically allocating per-vertex edge arrays becomes faster than radix sort when the storage medium is slow. On the SSD the total time for the radix sort approach is shorter than or more or less the same as the dynamic approach. The results for count sort are, as before, inferior, and are not included for that reason.

Pre-processing approach	RMAT26 out	RMAT26 in-out
Dynamic, loaded from SSD	20.7	40.0
Radix-sort, loaded from SSD	21.2	27.0
Dynamic, loaded from disk	61.0	61.1
Radix-sort, loaded from disk	65.0	71.0

Table 3: The cost of pre-processing for adjacency list creation with loading time included. Results show the time when building only the outgoing per-vertex edge arrays, and when building both the outgoing and incoming per-vertex edge arrays. The pre-processing is overlapped with loading when the adjacency list is created dynamically.

Summary. Costs associated with loading and building data structures in memory are non-negligible, and different approaches shine in different situations. Surprisingly, using radix sort to build adjacency lists is the fastest approach when the input file is in memory or loaded from a fast medium. When the graph is loaded from a slow medium, building adjacency lists dynamically is a better option, because it can be overlapped with loading.

4 Data layout and graph traversal

4.1 Vertex-centric vs. edge-centric

The choice of data layout impacts the decision of how to traverse the graph. In this section, we show that the best performing data layout and corresponding traversal model depend on the algorithm.

Computation on edge arrays happens in an **edge-centric** manner, and is quite simple: at every iteration of the computation the whole edge array is scanned, and the graph algorithm is called on every edge. This computation model is efficient, because scanning an edge array is cache-friendly: most of the accessed data is prefetched before being used. The drawback of this layout is that it offers no easy way to work on a subset of the vertices: a full scan of the edge array is required to find the edges of a vertex.

Adjacency lists are a natural solution to this problem. They enable **vertex-centric** computation, in which work is only performed on the subset of active vertices.

4.2 Evaluation

To illustrate the impact of data layout and traversal model on the end-to-end execution time, we show in Figure 3 the pre-processing and algorithm execution times of BFS, Pagerank, and SpMV on RMAT26. For BFS, vertex-centric computation performs the best, because during an iteration BFS only works on a limited subset of the graph. Edge arrays are not well suited for this type of computation, as all edges of the graph are read at every iteration.

In contrast, Pagerank accesses the entire graph in every iteration. Looking only at algorithm execution time, vertex-centric computation still performs a bit better, because it has better cache locality (all edges from a vertex are processed on the same core). When taking into account the pre-processing time, however, the end-to-end execution time is the same as for edge-centric computation.

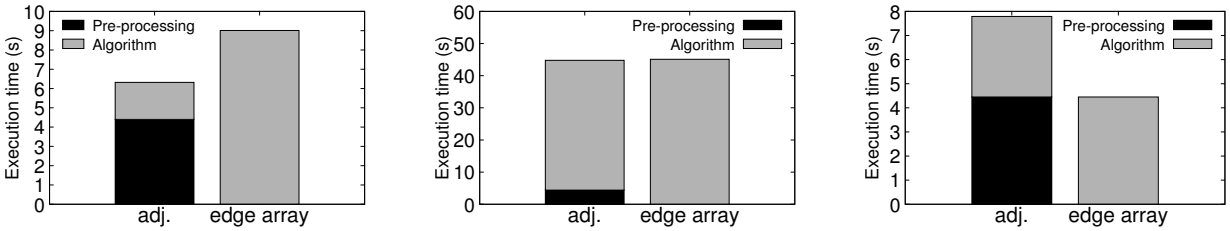
Finally, SpMV is an algorithm that makes only a single pass over the graph. Here, edge-centric computation produces the best end-to-end result, since the cost of building adjacency lists for vertex-centric execution is not amortized by any gains in algorithm execution time.

5 Cache-locality

Due to their irregular access patterns, graph algorithms usually exhibit poor cache locality. Last-level cache (LLC) misses may happen during three key steps of the computation: fetching an edge, fetching the metadata associated with the source vertex of the edge, and fetching the metadata associated with the destination vertex of the edge. In this section, we study how to lay out the data in memory to reduce the number of LLC misses, and we explain the pre-processing costs associated with creating those layouts.

5.1 Impact of the data layout

Edge array. In edge-centric computation, since edges are streamed, they are prefetched efficiently and do not incur cache misses. Fetching the metadata of the vertices, however, leads to random accesses with poor spatial and temporal locality.



(a) BFS on RMAT26 (b) Pagerank on RMAT26 (c) SPMV on RMAT26

Figure 3: Pre-processing and algorithm execution time for BFS, Pagerank and SpMV on RMAT26, using vertex-centric computation on an adjacency list or edge-centric computation on an edge array.

Adjacency lists. In adjacency lists, computation is performed from the point of view of a vertex: a core iterates over all edges of a given vertex before processing another vertex. As a consequence, the metadata of the source vertex is read only once, after which it is cached. This is beneficial for vertices that have a large number of edges. Fetching edges may introduce a cache miss for the first edge, but subsequent edges are prefetched, as with the edge array. Also similar to the case of the edge array, the metadata of the destination vertices exhibits poor cache behavior.

Grids: optimizing edge arrays. To improve the cache locality of edge arrays, data is laid-out as a grid of cells. Each cell contains the edges from a range of vertices to another range of vertices. Figure 4 shows an example of a graph transformed into a grid. This data structure is inspired by the grid data structure first introduced in GridGraph [37], which aimed at maximizing reuse of data read from disks. Computation then iterates over cells. The goal is that the metadata associated with the vertices in the cell stays in cache and can therefore be reused. We construct the grid using the same radix sort approach as for building adjacency lists. Instead of bucketing edges by source vertex, we bucket them by the cell to which they belong. The optimal number of cells in the grid depends on the graph shape and size. We experimentally find that a grid of 256x256 cells performs best on the Twitter and RMAT26 graphs. Building a grid is slightly more expensive than building an adjacency list (the number of cells in the grid is equal to $(\#vertices/256)^2$, which is higher than the number of vertices for large graphs).

We compare using radix sort with a dynamic approach for buiding the grid, and the conclusions regarding different pre-processing approaches made in Section 3.2 are applicable to grids as well: radix sort is faster when the graph is in memory or loaded from a fast medium, while dynamically building the grid is faster otherwise.

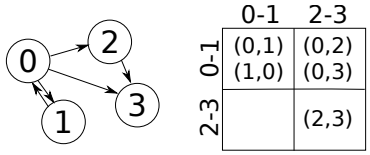
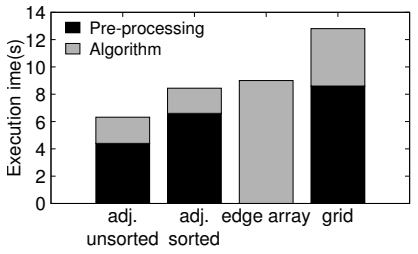


Figure 4: Transforming a graph into a grid representation.

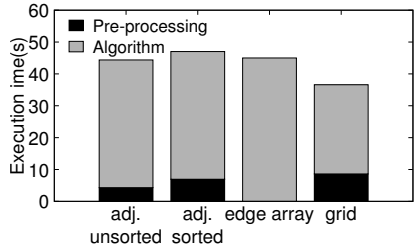
Optimizing adjacency lists. An intuitive idea to improve cache locality in adjacency lists is to sort the per-vertex edge arrays by destination. Indeed, the metadata of vertices with contiguous IDs is also contiguous in memory, thus when accessing vertex 0 and then vertex 1, the metadata of vertex 1 is likely to be present in cache. Of course, sorting the per-vertex edge arrays increases the pre-processing cost.

5.2 Evaluation

Figure 5 compares the pre-processing and algorithm execution times of BFS and Pagerank on RMAT26, on the unsorted adjacency list, the sorted adjacency list, the edge array and the grid. Table 4 presents the cache miss rate for these four data layouts.



(a) BFS on RMAT26



(b) Pagerank on RMAT26

Figure 5: Impact of cache-related optimizations on pre-processing and algorithm execution time for BFS and Pagerank on RMAT26.

Data layout	BFS	Pagerank
Edge array	57%	83%
Grid	23%	35%
Adjacency list	63%	78%
Adjacency list sorted	63%	78%

Table 4: Cache miss ratio for BFS and Pagerank on RMAT26.

BFS. For BFS, the unsorted adjacency list remains the solution with the best end-to-end execution time. Looking at algorithm execution time alone, BFS is 2.4× faster with a grid than with unsorted per-vertex edge arrays. However, creating the grid adds significant pre-processing time (9s), making the grid the slowest solution overall for BFS. Sorting the per-vertex edge arrays also leads to end-to-end performance inferior to unsorted adjacency lists. The pre-processing time increases, and the algorithm execution time does not decrease. Table 4 shows that sorting the per-vertex arrays does not significantly impact the cache miss rate. The destination vertices are accessed in order, but in practice a cache line only contains the metadata associated with very few vertices (64 in the case of BFS). Even when sorted, the destination vertex identifiers in the per-vertex edge arrays are sufficiently far apart for their metadata to fall in different cache lines, which explains the limited impact of this optimization on the number of cache misses and therefore on algorithm execution time. The increased pre-processing time for sorting the per-vertex arrays increases end-to-end execution time.

Pagerank. Even with the added pre-processing cost, the grid outperforms all other data layouts for Pagerank: it is 1.4× faster than an edge array and 1.3× faster than an unsorted adjacency list. This improvement is a direct result of the reduced cache miss rate when using a grid. As shown in Table 4, the cache miss ratio for the grid is less than half of that for the other data layouts. As for BFS, sorting the per-vertex edge arrays provides no benefit for Pagerank, for the same reasons. A cache line can fit at most 6 vertices for Pagerank, leading to an even smaller improvement in spatial locality than for BFS.

Summary. Creating a grid improves cache reuse and has a significant impact on algorithm execution time. Yet, this comes at the cost of an extra pre-processing, which is not always amortized. Different layouts also shine in very different situations. For instance, the grid is the best solution for Pagerank, but the slowest on BFS.

6 Information flow: Push and Pull

One of the core design decisions for a graph processing system is the information flow model it adopts. Information propagates through the graph in one of two ways: a vertex either **pushes** data along its out edges, writing to the state of its neighbors, or it **pulls** data along its incoming edges and updates its own state. These two approaches have important implications on computation, synchronization and pre-processing that we detail in this section.

6.1 Impact on end-to-end execution time

6.1.1 Impact on algorithm execution time

The **push** and **pull** approaches have different impact on the number of vertices and edges that need to be accessed during an iteration.

First, the **push** approach allows working on a subset of the vertices, while the **pull** approach does not. When pushing, vertices that do not need to propagate their value can be safely ignored. In contrast, the pull approach requires a vertex to scan all its incoming edges for neighbors that could potentially propagate a value. It also requires a pass over all vertices to check whether they need to look at their incoming edges (e.g., whether they have already been discovered in BFS).

Second, for some algorithms, the **pull** approach allows stopping the computation for a vertex in the middle of an iteration, while the **push** approach does not. Indeed, while **pulling** data a vertex may stop pulling before exploring all its incoming edges. For instance in BFS, if a vertex marks itself as discovered in the middle of an iteration, it stops exploring its remaining incoming edges. This guarantees that the vertex is discovered only once. In the **push** approach, vertices need to check that all their neighbors have been discovered, which leads to redundant work if multiple vertices have the same neighbors.

Figure 6 shows the per-iteration execution time of pushing vs. pulling for BFS on an RMAT26 graph. During the first iteration and after the third iteration, pushing is faster than pulling. During iterations 2 and 3, pulling is faster than pushing. This difference is explained by the percentage of the graph that is accessed during the iterations: most vertices in the graph are discovered during iterations 2 and 3. When pushing data, lots of redundant work is done in these iterations.

Because pushing data and pulling data perform best at different phases of the computation, some frameworks

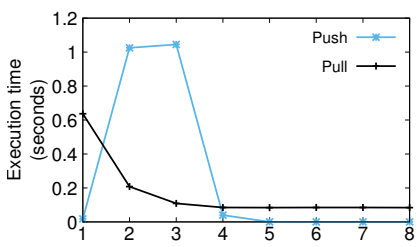


Figure 6: Per-iteration algorithm execution time for push vs. pull for BFS on RMAT26.

dynamically switch between pushing and pulling, depending on the number of active vertices in an iteration [2, 3, 29].

6.1.2 Impact on synchronization

A significant part of the algorithm execution time may involve synchronization. For example, in Pagerank on an RMAT26 graph with 16 cores, 40% of the algorithm execution time is spent in code protected by locks. The goal of this section is to evaluate the possibilities for lock removal, how they depend on the data layout and the information flow, and what if any pre-processing costs they induce.

In push mode, a vertex pushes updates to all its neighbors, and thus needs to lock them to update their metadata. In pull mode, a vertex only updates its own state. Thus, lock removal with adjacency lists requires execution in pull mode.

The grid offers a natural partition of the graph: edges in different rows have different source vertices, and edges in different columns have different destination vertices. To perform computation without locks in push mode, it suffices to assign different columns to different cores. To perform computation without locks in pull mode, it suffices to assign different rows to different cores.

6.1.3 Impact on pre-processing

Adjacency lists. To use push-pull, a system needs to iterate over both outgoing and incoming edges. As a result, when the graph is directed, we need to build both the outgoing and incoming per-vertex edge arrays. In contrast, for push we only need to build the outgoing, and for pull only the incoming per-vertex edge arrays. As a result, for directed graphs push-pull comes with an increased pre-processing cost, compared to push or pull, as seen in Section 3.2. When the graph is undirected, it suffices to build the outgoing per-vertex edge arrays (outgoing and incoming edges are the same), and push-pull induces no extra pre-processing cost.

Edge array. Computation over an edge array always requires scanning all the edges in the graph, so there is no advantage to using either push or pull. Furthermore, since the computation is edge-centric and not vertex-centric, locks need to be acquired for all updates. For these reasons, edge arrays are not considered any further in this section.

Lock removal. Lock removal does not require any additional pre-processing, beyond what is otherwise necessary for adjacency lists and grids, but it cannot be used with edge arrays, which have zero pre-processing cost.

6.2 Evaluation

6.2.1 BFS

Figure 7 presents the end-to-end execution times for BFS running on a directed RMAT26 graph, with adjacency lists, using push-pull, push (with locks) and pull (without locks). We do not show any results for edge array or grid for BFS, as we have shown in Section 5 that these approaches lead to inferior results compared to adjacency lists.

Push-pull is much faster in terms of algorithm execution time, but it is 1.5× slower than the push approach in terms of end-to-end execution time because of the extra pre-processing time. When taking pre-processing time into account, we find no combination of graphs, algorithms and machines in which push-pull is beneficial on directed graphs. On undirected graphs, push-pull does not add any pre-processing time, and is thus much faster than just pulling or pushing data. Furthermore, due to the fact that, on average, only a small percentage of vertices is processed per iteration, BFS in push mode performs 20% better than BFS in pull mode, even though push uses locks and pull does not.

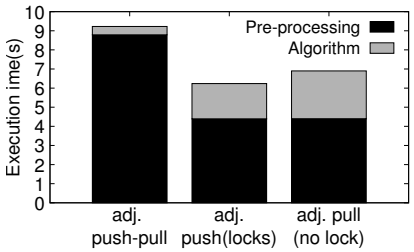


Figure 7: Pre-processing and algorithm execution time for BFS on RMAT26 using push-pull, push (with locks) and pull (without locks).

6.2.2 Pagerank

Figure 8 shows the end-to-end execution times for Pagerank in push mode on an adjacency list (with locks), in pull mode on an adjacency list (without locks), in push mode on a grid (with locks), and in pull mode on a grid (without locks). Here, the advantages of removing locks can be clearly seen. On adjacency lists, the version without locks is 40% faster than the push version when looking at end-to-end time. On a grid, the version without locks shows a gain of 1.5× in end-to-end time when comparing to the version with locks.

Summary. Push and pull on adjacency lists have conflicting benefits. Push works better for algorithms that only access a subset of the vertices in a given iteration, while pull allows vertices to be updated without locks. With grids, locking can be avoided regardless of whether push or pull is used, but the advantage of push remains

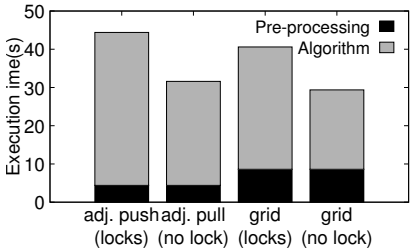


Figure 8: Pre-processing time and algorithm execution time for Pagerank on RMAT26 for push (with locks) on an adjacency list (with locks), for pull on an adjacency list (without locks), for push on a grid (with locks), for pull on a grid (without locks).

for algorithms that only access a subset of the vertices. Whether push or pull comes out ahead depends heavily on the nature of the algorithm. A combined push-pull approach requires extra pre-processing, which outweighs the benefits in terms of algorithm execution time.

7 NUMA-Awareness

We evaluate the trade-offs between the potential benefits of being NUMA-aware and the overheads it introduces in both the pre-processing and algorithm execution phase.

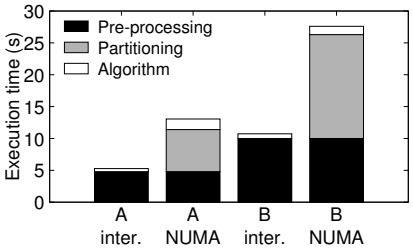
7.1 Data layout

In NUMA-aware solutions, the graph is *partitioned* across the NUMA nodes, and threads prioritize work from partitions that are local to their NUMA node. The partitioning scheme divides graph data evenly across NUMA nodes and places related data on the same NUMA node. Partitioning is performed so as to minimize the number of edges whose source and destination vertices are on different NUMA nodes, while still balancing the number of vertices and edges per NUMA node.

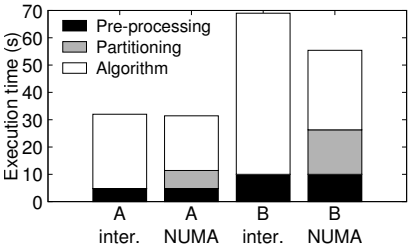
We evaluate in particular the partitioning schemes of Polymer [33] and Gemini [36]. The vertices are split into as many subsets as there are NUMA nodes. The outgoing edges of vertices are colocated with their target vertices. This approach avoids random remote writes and balances the number of edges across NUMA-nodes. Threads first process their local partitions. After that, they start working on remote partitions by updating the target vertices that are local to their NUMA node.

7.2 Evaluation

We evaluate the potential performance improvement of NUMA-aware data placement on the two machines presented in Section 2. Figure 9 shows the impact of NUMA-aware graph partitioning of an RMAT26 graph when running BFS and Pagerank. We compare NUMA partitioning to a solution that randomly interleaves the



(a) BFS - RMAT26



(b) Pagerank - RMAT26

Figure 9: Impact of NUMA-aware partitioning on machines A and B. For each machine we show the pre-processing, partitioning and algorithm execution time for BFS and Pagerank on RMAT26 with memory interleaving vs. NUMA-aware data placement.

graph data on all NUMA nodes. We use, for each application, the best algorithm in terms of algorithm execution, as presented in the previous sections (push/pull for BFS and pull without locks for Pagerank). The end-to-end execution time is broken down into pre-processing, partitioning and algorithm execution.

Looking at Figure 9b, the NUMA-aware data layout improves the algorithm execution time for Pagerank 1.3× on Machine A and 2× on Machine B. However, only on the machine B, with 4 NUMA nodes, does the end-to-end execution time benefit from being NUMA-aware.

In contrast, looking at Figure 9a, for BFS the NUMA-aware version is 3.5× slower on Machine A and 1.8× slower on Machine B. For BFS the time spent in partitioning dwarfs the algorithm execution time on both machines. More surprisingly, even when looking only at algorithm execution time, the NUMA-aware version performs worse than the interleaved version. In BFS, in a given iteration, only a small number of vertices is processed, and these vertices often share a common ancestor (e.g., during the first iteration, all processed vertices are the children of the root vertex). As a consequence, vertices processed during a given iteration often reside in the same partition. This leads to all cores accessing the same NUMA node, which creates memory contention [9]. This undesirable effect is even more visible on high-diameter graphs with low-degree vertices, as shown in Figure 10 when running BFS on the US-Road graph. The NUMA-aware version is 12× slower than the interleaved version.

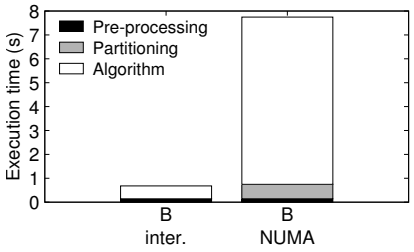


Figure 10: Effect of contention on memory bus on high diameter graphs. Pre-processing, partitioning and algorithm execution time for BFS US-Road graph with memory interleaving vs. NUMA-aware data placement

Summary. NUMA-aware data partitioning has a high pre-processing cost. This cost is amortized for algorithms that run for a long time and that work on most of the data during every iteration. For algorithms that run only for a short time, this may not be the case. For algorithms that only work on a subset of the data, NUMA-aware partitioning may exacerbate memory contention.

8 Additional algorithms and workloads

Table 5 shows the best solutions for BFS and Pagerank for graphs not evaluated in previous sections. The Twitter graph has a degree distribution similar to that of RMAT, and benefits from the same approaches: using an adjacency list while pushing data for BFS, and using a grid for Pagerank. The US-Road graph leads to slightly different conclusions. The best approach on Pagerank is to use an edge array and not a grid. Since the graph has a lower per-vertex degree than the RMAT and Twitter graphs, the grid data structure reduces only slightly the cache miss ratio, and therefore its pre-processing cost is not amortized.

In Table 6 we report the best approaches for WCC, SpMV, SSSP and ALS, their end-to-end execution time and its breakdown over pre-processing and algorithm execution time.

SPMV is a very short algorithm, and edge arrays are always the fastest approach, as they induce no pre-processing cost.

Intuitively, **WCC** should perform best on adjacency lists, because it is a traversal algorithm (only a subset of the graph is processed during every iteration of the computation), but WCC runs on an undirected graph. We therefore first have to build an undirected version of the graph from the input file. In the case of adjacency lists, an edge has to be inserted in both the outgoing edge array of its source and its destination. Thus, the pre-processing cost for creating adjacency lists is increased. In contrast, no additional pre-processing is required for edge arrays and grids to perform computation on an undirected graph. As a consequence, on graphs with a low diameter, WCC works best with an edge array, because the pre-processing time of building adjacency lists is too high. On graphs that have a higher diameter, like the US-Road graph, WCC needs more iterations to converge, and an adjacency list works best.

SSSP is very similar to BFS, and previous conclusions regarding the trade-offs between algorithm execution time and pre-processing for BFS are applicable to this algorithm as well. The only difference is that BFS discovers a vertex only once, whereas in SSSP a vertex may update its path many times during the computation, leading to an increase both in the number of iterations and the number of vertices active in each iteration.

ALS computes recommendations from a bipartite graph. The left side of the graph represents users and the other side items being rated. During every iteration, a subset of the graph (the left or right side) is active, and hence adjacency lists are the best data layout.

9 Summary

Improvements in algorithm execution time often come at the cost of increased pre-processing time. As seen in the previous sections, no approach fits every graph, algorithm or machine. In this section we try to provide a roadmap for choosing between different data layouts and computation approaches.

The first step consists of choosing an appropriate data layout. The layout is chosen based on the algorithm and graph characteristics. Short algorithms, such as SPMV, that complete in one iteration, should use an edge array, as it incurs no pre-processing cost. When the computation works only on a small subset of the graph at every computation step, adjacency lists in push mode improve algorithm execution time. The cost of building them is usually amortized compared to computation over edge arrays, especially on graphs with a high diameter. Other algorithms that run on graphs that have a large average per-vertex degree and iterate over most of the graph at every iteration, may benefit from using a grid, because the grid improves cache locality.

Second, if the machine is a large NUMA machine and the algorithm execution time is predicted to be large, then partitioning the graph to be NUMA-aware is beneficial (Figure 9b).

Third, if the data layout and computation approach chosen during the first step allow for execution without locking (e.g., pull mode in grids), then it is always beneficial to remove locks. We do not find any algorithm or directed graph for which switching between a pull mode without locks and push mode is beneficial when looking at end-to-end execution time.

Finally, when pre-processing cannot be avoided, it induces a non-negligible cost, and it should be optimized by using appropriate sorting techniques.

Algo	Graph	Data layout	Propagation model	Pre-processing	Algorithm	Total
BFS	Twitter	Adj. list	Push	5.8	2.3	8.1
BFS	US-Road	Adj. list	Push	0.3	0.5	0.8
Pagerank	Twitter	Grid	Pull (no lock)	23.2	37.8	61.0
Pagerank	US-Road	Edge array	Pull	0.0	1.6	1.6

Table 5: Best approaches in terms of end-to-end execution time for BFS and Pagerank on the Twitter and US-Road graph.

Algo	Graph	Data layout	Propagation model	Pre-processing	Algorithm	Total
WCC	RMAT-26	Edge array	Push	0.0	11.0	11.0
WCC	Twitter	Edge array	Push	0.0	19.2	19.2
WCC	US-Road	Adj. list	Push	0.6	56.8	57.4
SpMV	RMAT-26	Edge array	Push	0.0	4.4	4.4
SpMV	Twitter	Edge array	Push	0.0	5.8	5.8
SpMV	US-Road	Edge array	Push	0.0	0.3	0.3
SSSP	RMAT-26	Adj. list	Push	4.4	2.8	7.2
SSSP	Twitter	Adj. list	Push	5.8	4.4	10.2
SSSP	US-Road	Adj. list	Push	0.5	30.7	31.2
ALS	Netflix	Adj. list	Pull (no lock)	0.8	7.7	8.1

Table 6: Best approaches in terms of end-to-end execution time for SpMV, WCC and ALS on different graphs.

System	Data layout	Iteration model	Push or Pull	Without locks	NUMA-Aware
Ligra	Adj list	Vertex-centric	Push&Pull	Yes	-
Polymer	Adj list	Vertex-centric	Push&Pull	Yes	Yes
Gemini	Adj list	Vertex -centric	Push&Pull	Yes	Yes
X-Stream	Edge array	Edge-centric	Push	-	-
GridGraph	Grid	Grid-cell	Push	Yes	-

Table 7: Overview of multicore graph processing systems that inspired this work and their features.

10 Related Work

Very few papers compare the benefits of different graph processing systems. Satish et al. [28] evaluate various single-machine and distributed systems and compare them to a hand-optimized baseline. The paper looks at complete systems rather than individual techniques. Capota et al. [4] introduce a benchmark for graph processing platforms.

A large number of graph processing systems have been proposed [7, 8, 12–17, 19, 20, 22, 23, 25–27, 29, 31, 33, 36, 37]. We cover here only those works that have directly inspired the techniques evaluated in this paper. For a brief summary of the main features of these systems, see Table 7.

Beamer et al. [2, 3] are the first to propose push-pull for BFS. Ligra [29] extends this idea to other graph algorithms. It also uses radix sort for creating adjacency lists. X-Stream [27] introduces edge-centric graph processing in the context of out-of-core systems. GridGraph [37] improves on this idea by organizing the edges into a grid. Polymer [33] and Gemini [36] optimize graph processing for NUMA machines. We use their data placement technique in Section 7. In addition to the techniques used in Polymer, Gemini adds work stealing to balance work across NUMA nodes.

Not explored in this paper, the use of GPUs for graph processing has been the subject of some recent works [10, 11, 21, 30, 34]. This approach could affect the relative magnitude of pre-processing vs. algorithm execution time, and thereby impact the conclusions for certain algorithms.

11 Conclusion

We have presented an analysis of various techniques aimed at improving the algorithm execution time in graph processing systems, and we have explained their impact on pre-processing time. Our main observation is that pre-processing often dominates the end-to-end execution time of graph analytics. Therefore, it is often better to work with simple graph data layouts that induce less pre-processing than to invest time in elaborate pre-processing to speed up the algorithm execution phase. We argue that future works on graph analytics frameworks must more carefully consider this trade-off between pre-processing and algorithm execution time.

Acknowledgments: This work was supported in part by Swiss National Science Foundation Grant No. 167157 and by an EPFL-INRIA postdoctoral fellowship. We thank our reviewers, our shepherd Rong Chen, Laurent Bindschaedler, Florin Dinu, Rachid Guerraoui, Tim Harris, Dushyanth Narayanan, Amitabha Roy and Nicolas Schiper for their valuable feedback.

References

[1] http://dimacs.rutgers.edu/Challenges/.

[2] BEAMER, S., ASANOVIĆ, K., AND PATTERSON, D. Direction-optimizing breadth-first search. 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. 12:1–12:10.

[3] BEAMER, S., ASANOVIC, K., PATTERSON, D. A., BEAMER, S., AND PATTERSON, D. Searching for a parent instead of fighting over children: A fast breadth-first search implementation for graph500. Tech. rep.

[4] CAPOTĂ, M., HEGEMAN, T., IOSUP, A., PRAT-PÉREZ, A., ERLING, O., AND BONCZ, P. Graphalytics: A big data benchmark for graph-processing platforms. In *Proceedings of the GRADES'15* (New York, NY, USA, 2015), GRADES'15, ACM, pp. 7:1–7:6.

[5] CHAKRABARTI, D., ZHAN, Y., AND FALOUTSOS, C. R-MAT: A recursive model for graph mining. In *Proceedings of the SIAM International Conference on Data Mining* (2004), SIAM.

[6] 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. 15.

[7] CHENG, R., HONG, J., KYROLA, A., MIAO, Y., WENG, X., WU, M., YANG, F., ZHOU, L., ZHAO, F., AND CHEN, E. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the ACM European conference on Computer Systems* (2012), ACM, pp. 85–98.

[8] CHING, A. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara 11* (2011).

[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. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 381–394.

[10] DAVIDSON, A., BAXTER, S., GARLAND, M., AND OWENS, J. D. Work-efficient parallel GPU methods for single-source shortest paths. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium* (Washington, DC, USA, 2014), IPDPS '14, IEEE Computer Society, pp. 349–359.

[11] FU, Z., PERSONICK, M., AND THOMPSON, B. Mapgraph: A high level API for fast development of high performance graph analytics on GPUs. In *Proceedings of Workshop on Graph Data Management Experiences and Systems* (New York, NY, USA, 2014), GRADES'14, ACM, pp. 2:1–2:6.

[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 Conference on Operating Systems Design and Implementation* (2012), USENIX Association, pp. 17–30.

[13] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI'14, USENIX Association, pp. 599–613.

[14] HAN, W., MIAO, Y., LI, K., WU, M., YANG, F., ZHOU, L., PRABHAKARAN, V., CHEN, W., AND CHEN, E. Chronos: A graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 1:1–1:14.

[15] HONG, S., CHAFI, H., SEDLAR, E., AND OLUKOTUN, K. Green-Marl: A DSL for easy and efficient graph analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS XVII, ACM, pp. 349–362.

[16] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 59–72.

[17] JU, W., LI, J., YU, W., AND ZHANG, R. iGraph: an incremental data processing system for dynamic graph. *Frontiers of Computer Science* (2016), 1–15.

[18] KWAK, H., LEE, C., PARK, H., AND MOON, S. What is Twitter, a social network or a news media? In *Proceedings of the International conference on World Wide Web* (2010), ACM, pp. 591–600.

[19] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Imple-*

mentation (Berkeley, CA, USA, 2012), USENIX Association, pp. 31–46.

[20] 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 Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 527–543.

[21] MERRILL, D., GARLAND, M., AND GRIMSHAW, A. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2012), PPoPP '12, ACM, pp. 117–128.

[22] 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* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 439–455.

[23] NGUYEN, D., LENHARTH, A., AND PINGALI, K. A lightweight infrastructure for graph analytics. In *Proceedings of the Symposium on Operating Systems Principles* (2013), ACM, pp. 456–471.

[24] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[25] PEREZ, Y., SOSIČ, R., BANERJEE, A., PUTTAGUNTA, R., RAISON, M., SHAH, P., AND LESKOVEC, J. Ringo: Interactive graph analytics on big-memory machines. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2015), SIGMOD '15, ACM, pp. 1105–1110.

[26] 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), ACM, pp. 410–424.

[27] ROY, A., MIHAILOVIC, I., AND ZWAENEPOEL, W. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the ACM symposium on Operating Systems Principles* (2013), ACM, pp. 472–488.

[28] SATISH, N., SUNDARAM, N., PATWARY, M. M. A., SEO, J., PARK, J., HASSAAN, M. A., SENGUPTA, S., YIN, Z., AND DUBEY, P. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, ACM, pp. 979–990.

[29] 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.

[30] 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 *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2016), PPoPP '16, ACM, pp. 11:1–11:12.

[31] 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 Sixth ACM Symposium on Cloud Computing* (New York, NY, USA, 2015), SoCC '15, ACM, pp. 408–421.

[32] ZAGHA, M., AND BLELLOCH, G. E. Radix sort for vector multiprocessors. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing* (1991), ACM, pp. 712–721.

[33] ZHANG, K., CHEN, R., AND CHEN, H. NUMA-aware graph-structured analytics. In *ACM SIGPLAN Notices* (2015), vol. 50, ACM, pp. 183–193.

[34] ZHONG, J., AND HE, B. Medusa: Simplified graph processing on GPUs. *IEEE Trans. Parallel Distrib. Syst. 25*, 6 (June 2014), 1543–1552.

[35] ZHOU, Y., WILKINSON, D., SCHREIBER, R., AND PAN, R. Large-scale parallel collaborative filtering for the Netflix Prize. In *Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management* (Berlin, Heidelberg, 2008), AAIM '08, Springer-Verlag, pp. 337–348.

[36] ZHU, X., CHEN, W., ZHENG, W., AND MA, X. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)(Savannah, GA* (2016).

[37] 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), pp. 375–386.

Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX

Chia-Che Tsai
Stony Brook University

Donald E. Porter
University of North Carolina at Chapel Hill and Fortanix

Mona Vij
Intel Corporation

Abstract

Intel SGX hardware enables applications to protect themselves from potentially-malicious OSes or hypervisors. In cloud computing and other systems, many users and applications could benefit from SGX. Unfortunately, current applications will not work out-of-the-box on SGX. Although previous work has shown that a library OS can execute unmodified applications on SGX, a belief has developed that a library OS will be ruinous for performance and TCB size, making application code modification an implicit prerequisite to adopting SGX.

This paper demonstrates that these concerns are exaggerated, and that a fully-featured library OS can rapidly deploy unmodified applications on SGX with overheads comparable to applications modified to use "shim" layers. We present a port of Graphene to SGX, as well as a number of improvements to make the security benefits of SGX more usable, such as integrity support for dynamically-loaded libraries, and secure multi-process support. Graphene-SGX supports a wide range of unmodified applications, including Apache, GCC, and the R interpreter. The performance overheads of Graphene-SGX range from matching a Linux process to less than 2× in most single-process cases; these overheads are largely attributable to current SGX hardware or missed opportunities to optimize Graphene internals, and are not necessarily fundamental to leaving the application unmodified. Graphene-SGX is open-source and has been used concurrently by other groups for SGX research.

1 Introduction

Intel SGX introduces a number of essential hardware features that allow an application to protect itself from the host OS, hypervisor, BIOS, and other software. With SGX, part or all of an application can run in an *enclave*. Enclave features include confidentiality and integrity protection for the enclave's virtual address space; restricting control flow into well-defined entry points for an enclave; integrity checking memory contents at start time; and remote attestation. SGX is particularly appealing in cloud computing, as users might not fully trust the cloud provider. That said, for any sufficiently-sensitive application, using SGX may be prudent, even within one administrative domain, as the security track record of commodity operating systems is not without blemish. Thus, a significant number of users would benefit from running applications on SGX as soon as possible.

Unfortunately, applications do not "just work" on SGX. SGX imposes a number of restrictions on enclave code that require application changes or a layer of indirection. Some of these restrictions are motivated by security, such as disallowing system calls inside of an enclave, so that system call results can be sanitized by *shielding code* in the enclave before use. Our experience with supporting a rich array of applications on SGX, including web servers, language runtimes, and command-line programs, is that a number of software components, orthogonal to the primary functionality of the application, rely on faithful emulation of arcane Linux system call semantics, such as `mmap` and `futex`; any SGX wrapper library must either reproduce these semantics, or large swaths of code unrelated to security must be replaced. Although this paper focuses on SGX, we note that a number of vendors are developing similar, but not identical, hardware protection mechanisms, including IBM's SecureBlue++ [16] and AMD SEV [27]—each with different idiosyncrasies. Thus, the need to adapt applications to use hardware security features will only increase in the near term.

As a result, there is an increasingly widespread belief that adopting SGX necessarily involves significant code changes to applications. Although Haven [15] showed that a library OS could run unmodified applications on SGX, this work pre-dated availability of SGX hardware. Since then, several papers have argued that the library OS approach is impractical for SGX, both in performance overhead and trusted computing base (TCB) bloat, and that one must instead refactor one's application for SGX. For instance, a feasibility analysis in the SCONE paper concludes that "On average, the library OS increases the TCB size by 5×, the service latency by 4×, and halves the service throughput" [14]. Shinde et al. [49] argue that using a library OS, including libc, increases TCB size by two orders of magnitude over a thin wrapper.

This paper demonstrates that these concerns are greatly exaggerated: one can use a library OS to quickly deploy applications in SGX, gaining immediate security benefits without crippling performance cost or TCB

bloat. We present a port of the Graphene library OS [52] to SGX, called Graphene-SGX, and show that the performance overheads are comparable to the range of overheads presented in SCONE; the authors of Panoply also note that Graphene-SGX is actually 5-10% faster than Panoply [49]. Arguments about TCB size are more nuanced, and a significant amount of the discrepancies arise when comparing incidental choices like libc implementation (e.g., musl vs. glibc). Graphene, not including libc, adds 53 kLoC to the application's TCB, which is comparable to Panoply's 20 kLoC or SCONE's 97 kLoC. Our position is that the primary reduction to TCB comes from either compiling out unused library functionality, as in a unikernel [38] and measured by our prior work [53]; or further partitioning an application into multiple enclaves with fewer OS requirements. When one normalizes for functionality required by the code in the enclave, the design choice between a library OS or a smaller shim does not have a significant impact on TCB size.

To be clear, SGX-specific coding has benefits, but we must not let the perfect be the enemy of the good. For example, privilege separating a complex application into multiple enclaves may be a good idea for security [40, 44, 49], and replacing particularly expensive operations can improve performance on SGX. The goal of Graphene is to bring up rich applications on SGX quickly, and then let developers optimize code or reduce the TCB as needed.

Graphene-SGX runs unmodified Linux binaries on SGX; to this end, this paper also contributes a number of usability enhancements, including integrity support for dynamically-loaded libraries, enclave-level forking, and secure inter-process communication (IPC). Users need only configure features and cryptographically sign the configuration.

Graphene-SGX is also useful as a tool to accelerate SGX research. Graphene-SGX has been open-sourced since June 2016[1]. Although our focus is unmodified applications, Graphene-SGX can also run smaller pieces of code in an enclave, as in a partitioned application. Several papers already compared against or extended Graphene-SGX [28, 43, 49] and we are aware of ongoing projects using Graphene-SGX.

The contributions of this paper are:

- A framework, called Graphene-SGX, to isolate unmodified, Linux applications in enclaves.
- Several usability enhancements for SGX, including dynamic loading, `fork`, and IPC.
- A thorough evaluation of the performance of unmodified applications on Graphene-SGX, indicating that the costs of a feature-rich library OS on SGX are in-band with purportedly lighter-weight solutions that require application changes. For example, lighttpd throughput and latency on Graphene-SGX are comparable to a Linux process. Overheads are generally under 2× (cf. SCONE overheads up to 1.6× on comparable workloads). In a few cases, Graphene-SGX overheads are higher, but these are internal to the library OS or fundamental to enclave limitations, not because the application is unmodified.

[1] Available at https://github.com/oscarlab/graphene

2 Background

This section summarizes SGX, and current design points for running or porting applications on SGX.

2.1 Software Guard Extensions (SGX)

The primary SGX abstraction is an *enclave*: an isolated execution environment within the virtual address space of a process. The code and data in enclave memory do not leave the CPU package unencrypted; when memory contents are read back into cache, the CPU decrypts the contents, and checks the integrity of cache lines and the virtual-to-physical mapping. SGX also cryptographically measures the integrity of enclaves at start-up, and provide attestation to remote systems or other enclaves.

SGX enables a threat model where one only trusts the Intel CPUs and the code running in the enclave(s). SGX protects applications from three different types of attacks on the same host, which are summarized in Figure 1: untrusted application code inside the same process but outside the enclave; operating systems, hypervisors, and other system software; other applications on the same host; and off-chip hardware. A SGX enclave can also trust a remote service or enclave, and be trusted after inter-platform attestation [13].

2.2 SGX Software Design Space

This subsection summarizes the principal design choices facing any framework for running applications on SGX. We explain the decisions in recent systems for SGX applications, and the trade-offs in this space.

How much functionality to pull into the enclave? At one extreme, a library OS like Haven [15] pulls most of the application-supporting code of the OS into the enclave. On the other extreme, thin "shim" layers, like SCONE [14] and Panoply [49] wrap an API layer such as the system call table. Pulling more code into the enclave increases the size of the TCB, but can reduce the size and complexity of the interface, and attack surface, between the enclave and the untrusted OS.

The impact of this choice on performance largely depends on two issues. First, entering or exiting the enclave is expensive; if the division of labor reduces enclave border crossings, it will improve performance. The second is the size of the Enclave Page Cache (EPC), limited to 128MB on version 1 of SGX. If a large support-

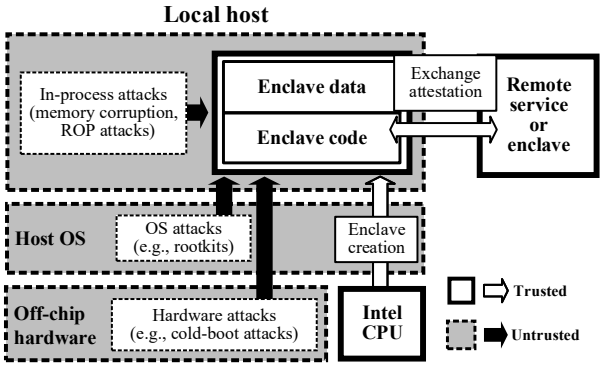


Figure 1: The threat model of SGX. SGX protects applications from three types of attacks: in-process attacks from outside of the enclave, attacks from OS or hypervisor, and attacks from off-chip hardware.

ing framework tips the application's working set size past this mark, the enclave will incur expensive swapping.

Shielding complexity. SGX hardware can isolate an application from an untrusted OS, but SGX alone can't protect an application that requires functionality from the OS. *Iago attacks* [18] are semantic attacks from the untrusted OS against the application, where an unchecked system call return value or effect compromises the application. Iago attacks can be subtle and hard to comprehensively detect, at least with the current POSIX or Linux system call table interfaces.

Thus, any SGX framework must provide some *shielding* support, to validate or reject inputs from the untrusted OS. The complexity of shielding is directly related to the interface complexity: inasmuch as a library OS or shim can reduce the size or complexity of the enclave API, the risks of a successful Iago attack are reduced.

Application code complexity. Common example applications for SGX in the literature amount to a simple network service running a TLS library in the enclave, putting minimal demands on a shim layer. Even modestly complex applications, such as the R runtime and a simple analytics package, require dozens of system calls providing wide-ranging functionality, including `fork` and `execve`. For these applications, the options for the user or developer become: (1) modifying the application to require less of the runtime; (2) opening and shielding more interfaces to the untrusted OS; (3) pulling more functionality into a shim or a library OS. The goal of this paper is to provide an efficient baseline, based on (3), so that users can quickly run applications on SGX, and developers can explore (1) or (2) at their leisure.

Application partitioning. An application can have multiple enclaves, or put less important functionality outside of the enclave. For instance, a web server can keep cryptographic keys in an enclave, but still allow client requests to be serviced outside of the enclave. Similarly, a privilege-separated or multi-principal application might create a separate enclave for each privilege level.

This level of analysis is application-specific, and beyond the focus of this paper. However, partitioning a complex application into multiple enclaves can be good for security. In support of this goal, Graphene-SGX can run smaller pieces of code, such as a library, in an enclave, as well as coordinate shared state across enclaves.

3 Design Overview

This section discusses the threat model, how Graphene-SGX defends against attacks from the untrusted OS, and how users configure policies for defenses.

3.1 Threat Model

Graphene-SGX follows a typical threat model for SGX applications. The following components are untrusted: (1) hardware outside of the Intel CPU package(s), (2) the OS, hypervisor, and other system software, (3) other applications executing on the same host, including unrelated enclaves, and (4) user-space components that reside in the application process but outside the enclave. Our design only trusts the CPUs and any code running inside the enclave, including the library OS, the unmodified application, and its supporting libraries.

We also trust `aesmd`, an enclave provided by the Intel's SGX SDK, which verifies attributes in the enclave signature and approves the enclave creation. Currently, any framework that uses SGX for remote attestation must trust `aesmd`. Graphene-SGX uses, but does not trust, the Intel SGX kernel driver. Other than `aesmd` and the driver, Graphene-SGX does not use or trust any part of the SDK.

Denial of service, side channels, and controlled-channel attacks [54] are vulnerabilities common to all SGX frameworks, and are beyond the scope of this work.

3.2 User Policy Configuration

Before an application is first executed using Graphene-SGX, the user must make certain policy decisions. Our goal is to balance policy expressiveness with usability.

As with Graphene and several other systems, each application requires a *manifest* to specify which resources the application is allowed to use, including a unioned, chroot-style view of the file system (comparable to aufs), and a set of iptables-style network rules. In Graphene, a program cannot access any resources not declared in the manifest. The original intention of the manifest was to protect the host: a reference monitor can easily identify the resources an application might use, and reject an application with a problematic manifest.

In Graphene-SGX, the manifest is extended to protect the application from the host file system. Specifically,

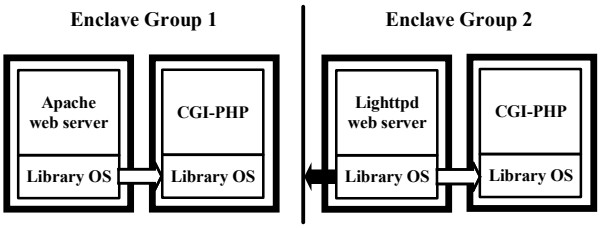


Figure 2: Two enclave groups, one running Apache and the other running Lighttpd, each creates a child enclave running CGI-PHP. Graphene-SGX distinguishes the child enclaves in different enclave groups.

the manifest can specify secure hashes (using SHA-256) of trusted files (generally read-only, including dynamic libraries). As part of opening a file, Graphene-SGX verifies the integrity of trusted files by checking the secure hashes. A trusted file is only opened if the secure hash matches. The manifest can also specify files or directories that can be accessed but are not trusted, such as a write-only output file. Graphene-SGX includes a signing utility that hashes all trusted files and generates a signed manifest that can be used at runtime.

SGX requires that certain resources be specified at initialization time, including the number of threads, the maximum size of the enclave, and the starting virtual address of the enclave. Thus, we also extend the manifest syntax for the user to specify these options. Other security-sensitive manifest options inherited from Graphene, such as enabling debug output, are also protected as part of the signed manifest.

3.3 Multi-Process Applications

Graphene supports multi-process applications by running a separate library OS instance in each process [52]. Each library OS instance coordinates state via message passing. Graphene implements Linux multi-process abstractions in the user-space, including `fork`, `execve`, signals, and System V semaphores and message queues.

Graphene-SGX extends the multi-process support of Graphene to enclaves by running each process with a library OS instance in an enclave. For instance, `fork` creates a second enclave and copies the parent enclave's contents over message passing. We call a group of coordinating enclaves an *enclave group*. Figure 2 shows two mutually-untrusting enclave groups running on a host.

Because multi-process abstractions are implemented in enclaves, securing these abstractions from the OS is straightforward. Graphene-SGX adds: (1) the ability for enclaves to authenticate each other via local attestation, and thereby establish a secure channel, and (2) a mechanism to securely fork into a new enclave, adding the child to the enclave group (§4.3).

4 Shielding Linux Abstractions

This section discusses how Graphene-SGX implements and shields the Linux ABI for applications in enclaves.

4.1 Shielding Dynamic Loading

To run unmodified Linux binaries, Graphene-SGX implements dynamic loading and run-time linking. In a major Linux distribution like Ubuntu, more than 99% of binaries are dynamically linked [53]. Static linking is popular for SGX frameworks because it is simple and facilitates the use of hardware enclave measurements. Dynamic linking requires rooting trust in a dynamic loader, which must then measure the libraries. For Haven [15], the enclave measurement only verifies the integrity of Haven itself, and the same measurement applies to any application running on the same Haven binary.

Graphene-SGX extends the Haven model to generate a unique signature for any combination of executable and dynamically-linked libraries. Figure 3 shows the architecture and the dynamic-loading process of an enclave. Graphene-SGX starts with an untrusted Platform Adaption Layer (`pal-sgx`), which calls the SGX drivers to initialize the enclave. The initial state of an enclave, which determines the measurement then attested by the CPU, includes a shielding library (`libshield.so`), the executable to run, and a manifest file that specifies the attributes and loadable binaries in this enclave. The shielding library then loads a Linux library OS (`libLinux.so`) and the standard C libraries (`ld-linux-x86-64.so` and `libc.so`). After enclave initialization, the loader continues loading additional libraries, which are checked by the shielding libraries. If the SHA-256 hash does not match the manifest, the shield will refuse to open the libraries.

To reiterate, a manifest includes integrity measurements of all components and is signed; this manifest is unique for each application and is measured as part of enclave initialization. This strategy does require trust in the Graphene (in-enclave) bootloader and shielding module to correctly load binaries according to the manifest and reject any errant binaries offered by the OS. This is no worse than the level trust placed in Haven's dynamic loader, but differentiates applications or even instances of the same application with different libraries.

Memory permissions. By default, the Linux linker format (ELF) often places code and linking data (e.g., jump targets) in the same page. It is common for a library to temporarily mark an executable pages as writable during linking, and then protect the page to be execute-only. This behavior is ubiquitous in current Linux shared libraries, but could be changed at compile time to pad writable sections onto separate pages.

The challenge on version 1 of SGX is that an application cannot revoke page permissions after the enclave

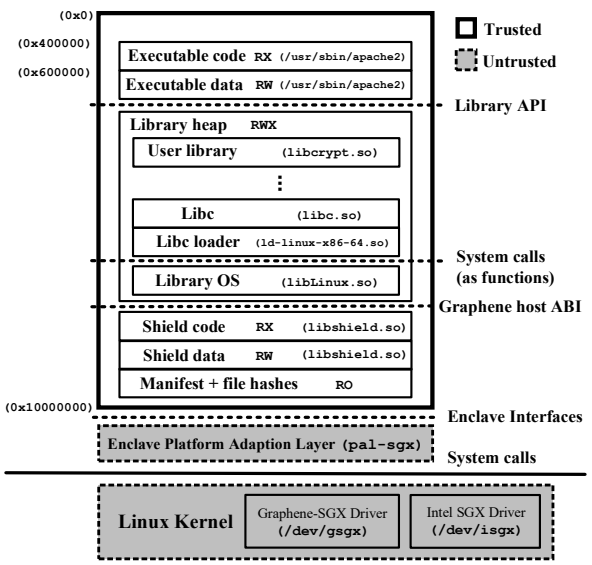


Figure 3: The Graphene-SGX architecture. The executable is position-dependent. The enclave includes an OS shield, a library OS, libc, and other user binaries.

starts. In order to support this ELF behavior, we currently map all enclave pages as readable, writable, and executable. This can lead to some security risks, such as code injection attacks in the enclave. In a few cases, this can also harm functionality; for instance, some Java VM implementations use page faults to synchronize threads. Version 2 of SGX [41] will support changing page protections, which Graphene-SGX will adopt in the future.

Position-dependent executables. SGX requires that all enclave sizes be a power-of-two, and that the enclave starts at a virtual address aligned to the enclave size. Most Ubuntu Linux executables are compiled to be position-dependent, and typically start at address `0x400000`. The challenge is that, to create an enclave that includes this address and is larger than 4MB, the enclave will necessarily need to include address zero.

We see including address zero in the enclave as a net positive, but not strictly necessary, as we are reluctant to make strong claims in the presence of code that follows null pointers. Graphene-SGX can still mark this address as unmapped in an enclave. Thus, a null pointer will still result in a page fault. On the other hand, if address zero were outside of the enclave, there is a risk that the untrusted OS could map this address to dangerous data [10], undermining the integrity of the enclave.

4.2 Shielding Single-Process Abstractions

For a single-process application running on Graphene-SGX, most Linux system calls are serviced inside the enclave by the library OS. A Graphene-SGX enclave includes both the same library OS in "classic" Graphene,

Classes	**Safe**	Benign	DoS	Unsafe
Enter enclaves & threads	**2**	0	0	0
Clone enclaves & threads	**2**	0	0	0
File & directory access	**3**	0	0	2
Exit enclave	**1**	0	0	0
Network & RPC streams	**5**	2	0	0
Scheduling	**0**	1	1	0
Stream handles	**2**	2	1	0
Map untrusted memory	**2**	0	0	0
Miscellaneous	**1**	1	0	0
Total	**18**	6	2	2

Table 1: 28 enclave interfaces, including *safe* (host behavior can be checked), *benign* (no harmful effects), *DoS* (may cause denial-of-service), and *unsafe* (potentially attacked by the host) interfaces.

that would also run on a Linux or FreeBSD picoprocess, as well as an SGX-specific platform adaptation layer (PAL), which implements 36 functions of the host ABI that the library OS is programmed against. This PAL funnels to a slightly smaller set of 28 interfaces which the enclave calls out to the untrusted OS (Table 1).

The evolution of the POSIX API and Linux system call table were not driven by a model of mutual distrust, and retrofitting protection onto this interface is challenging. Checkoway and Shachman [18] demonstrate the subtlety of detecting semantic attacks via the POSIX interface. Projects such as Sego [33] go to significant lengths, including modifying the untrusted OS, to validate OS behavior on subtle and idiosyncratic system calls, such as `mmap` or `getpid`.

The challenge in shielding an enclave interface is carefully defining the expected behavior of the untrusted system, and either validating the responses, or reasoning that any response cannot harm the application. By adding a layer of indirection under the library OS, we can define an enclave ABI that has more predictable semantics, which is, in turn, more easily checked at run-time. For instance, to read a file, Graphene-SGX requests that untrusted OS to map the file at an address outside the enclave, starting at an absolute offset in the file, with the exact size that the library OS needs for checking. After copying chunks of the file into the enclave, but before use, the contents can be hashed and checked against the manifest. This enclave interface limits the possible return values to one predictable answer, and thus reduces the space that the OS can explore to find attack vectors to the enclave. Many system calls are partially (e.g., `brk`) or wholly (e.g., `fcntl`), absorbed into the library OS, and do not need shielding from the untrusted OS.

Table 1 lists our 28 enclave interfaces, organized by risk. 18 interfaces are *safe* because responses from the OS are easily checked in the enclave. An example of a safe interface is `FILE_MAP`, which maps a file outside

the enclave, to copy it into the enclave for system calls like `mmap` or `read`, as discussed below. 6 interfaces are *benign*, which means, if a host violates the specification, the library OS can easily compensate or reject the response. An example of a benign interface is `STREAM_FLUSH`, which requests that data be sent over a network or to disk; cryptographic integrity checks on a file or network communication can detect when this operation is ignored by untrusted software.

Like any SGX framework, Graphene-SGX does not guarantee liveness of enclave code: the OS can refuse to schedule the enclave threads. Two interfaces are susceptible to liveness issues (labeled *DoS*): `FUTEX_WAIT` and `STREAM_POLL`. In the example of `FUTEX_WAIT`, a blocking synchronization call may never return, violating liveness but not safety. A malicious OS could cause a futex wait to return prematurely; thus, synchronization code in the PAL must handle spurious wake-ups and either attempt to wait on the futex again, or spin in the enclave.

Finally, only two interfaces, namely `FILE_STAT` and `DIR_READ`, are *unsafe*, because we do not protect integrity of file metadata. We leave this issue for future work, adopting one of several existing solutions [21].

File authentication. As with libraries and application binaries, configuration files and other integrity-sensitive data files can have SHA256 hashes listed in the signed manifest. At the first `open` to ones of the listed files, Graphene-SGX maps the whole file outside the enclave, copies the content in the enclave, divides into 64KB chunks, constructs a Merkle tree of the chunk hashes, and finally validates the whole-file hash against the manifest. In order to reduce enclave memory usage, Graphene-SGX does not cache the whole file after validating the hash, but keeps the Merkle tree to validate the untrusted input for subsequent, chunked reads. The Merkle tree is calculated using AES-128-GMAC.

Memory mappings. The current SGX hardware requires that the maximum enclave size be set at creation time. Thus, a Graphene-SGX manifest can specify how much heap space to reserve for the application, so that the enclave is sufficiently large. This heap space is also used to cache file contents.

Threading. Graphene-SGX currently uses a 1:1 threading model, whereas SCONE and Panoply support an m:n threading model. The issue is that SGX version 1 requires the maximum number of threads in the enclave to be specified at initialization time. We see this as a short-term problem, as SGX version 2 will support dynamic thread creation. We currently have users specify how many threads the application needs in the manifest.

This choice affect performance, as one may be able to use m:n threading and asynchronous calls at the enclave boundary to reduce the number of exits. This is a good idea we will probably implement in the future. Eleos [43] addresses this performance problem on unmodified Graphene-SGX with application-level changes to issue asynchronous system calls. The benefits of this optimization will probably be most clear in I/O-bound network services that receive many concurrent requests.

Exception handling. Graphene-SGX handles hardware exceptions triggered by memory faults, arithmetic errors, or illegal instructions in applications or the library OS. SGX does not allow exceptions to be delivered directly into the enclave. An exception interrupts enclave execution, saves register state on a thread-specific stack in the enclave, and returns to the untrusted OS. When SGX re-enters the enclave, the interrupted register state is then used by Graphene-SGX to reconstruct the exception, pass it to the library OS, and eventually deliver a signal to the application.

We note that the untrusted OS may deliberately trigger memory faults, by modifying the page tables, or not deliver the exceptions (denial of service). Direct exception delivery within an enclave is an opportunity to improve performance and security in future generations of SGX, as designed in Sanctum [19].

By handling exceptions inside the enclave, Graphene-SGXcan emulate instructions that are not supported by SGX, including `cpuid` and `rdtsc`. Use of these instructions will ultimately trap to a handler inside the enclave, to call out to the OS for actual values, which are treated as untrusted input and are checked.

4.3 Shielding Multi-Process Abstractions

Many Linux applications use multi-process abstractions, which are implemented using copy-on-write fork and in-kernel IPC abstractions. In SGX, the host OS is untrusted, and enclaves cannot share protected memory. Fortunately, Graphene implements multi-process support including `fork`, `execve`, signals, and a subset of System V IPC, using message passing instead of shared memory. Thus, Graphene-SGX implements multi-process abstractions in enclaves without major library OS changes. This subsection explains how Graphene-SGX protects multi-processing abstractions from an untrusted OS.

Process creation in Graphene-SGX is illustrated in Figure 4. When a process in Graphene-SGX forks into a new enclave, the parent and child will be running the same manifest and binaries, and will have the same measurements. Similar to the process creation in Graphene, the parent and child enclaves are connected with a pipe-like RPC stream, through the untrusted PAL. As part of initialization, the parent and child will exchange a session key over the unsecured RPC stream, using Diffie-Hellman. The parent and child use the CPU to generate attestation reports, which include a 512-bit field in the report to store a hash of the session key and a unique en-

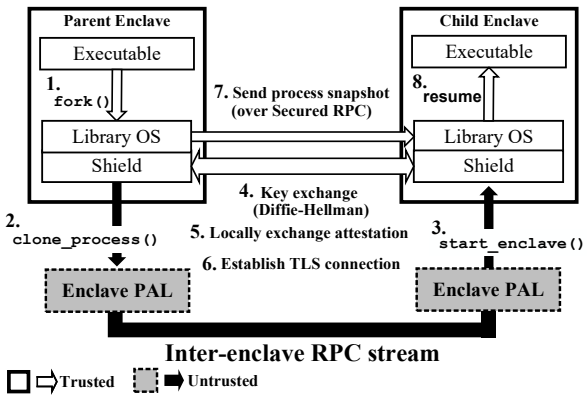


Figure 4: Process creation in Graphene-SGX. Numbers show the order of operations. When a process forks, Graphene-SGX creates a new, clean enclave on the untrusted host. Then the two enclaves exchange an encryption key, validates the CPU-generated attestation of each other, and migrates the parent process snapshot.

clave ID. The parent and child exchange these reports to authenticate each other. Unlike remote attestation, local attestation does not require use of Intel's authentication service (IAS). Once the parent and child have authenticated each other, the parent establishes a TLS connection over the RPC stream using the session key. The parent can then send a snapshot of itself over the TLS-secured RPC stream, and the snapshot is resumed in the child process, making it a clone of its parent. This mutual attestation and encryption strategy prevents a man-in-the-middle attack between the parent and child.

Once a parent enclave forks a child, by default, the child is fully trusted. To create a less trusted child, the parent would need to sanitize its snapshot, similar in spirit to the close-on-exec flag for file handles. For example, a pre-forked Apache web server may want to keep worker processes isolated from the master to limit a potential compromise of a worker process. Graphene-SGX inherits a limited API from Graphene, for applications to isolate themselves from untrusted child processes, but developers are responsible for purging confidential information before isolation.

Supporting `execve`. Unlike `fork`, `execve` starts a process with a specific executable, often different from the caller. When a thread calls `execve` in Graphene-SGX, the library OS migrates the thread to a new process, with file handles being inherited. Although the child does not inherit a snapshot from its parent, it can still compromise the parent by exploiting potential vulnerabilities in handling RPC, which are not internally shielded. In other words, Graphene-SGX is not designed to share library OS-internal with untrusted children. Thus, Graphene-SGX restricts `execve` to only launch trusted executables, which are specified in the manifest.

Inter-process communication. After process creation, parent and child processes will cooperate through shared abstractions, such as signals or System V message queues, via RPC messages. While messages are being exchanged between enclaves, they are encrypted, ensuring that these abstraction are protected from the OS.

5 Evaluation

Graphene-SGX is designed to be general-purpose, supporting a broad range of server and command-line applications. We thus evaluate performance overheads of unmodified Linux applications, using binaries from an Ubuntu installation. Depending on the workload, we measure application throughput or latency.

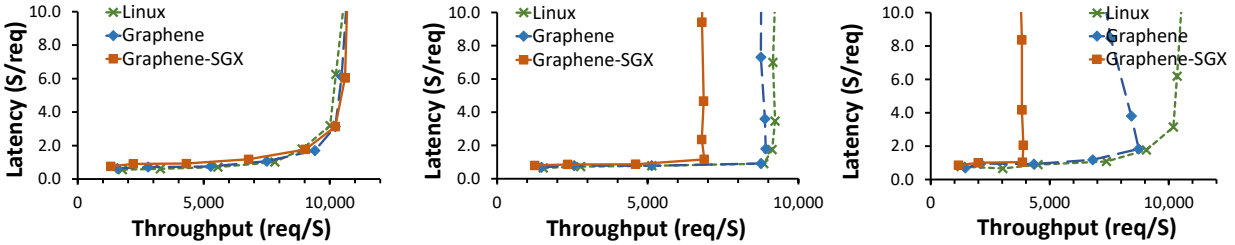
In order to differentiate SGX-specific overheads from Graphene overheads, we use both Linux processes and Graphene on a Linux host without SGX as baselines for comparison. Note that Graphene includes two optional kernel extensions: one that creates a reference monitor to protect the host kernel from the library OS, and one that optimizes fork by with copy-on-write for large (page-sized) RPC messages. Neither of these extensions are currently supported in Graphene-SGX.

Experimental setup. We use a Dell Optiplex 790 Small-Form Desktop, with a 4-core 3.20 GHz Intel Core i5-6500 CPU (no hyper-threading, with 6MB cache), 8 GB RAM, and a 512GB, 7200 RPM SATA disk. The host OS is Ubuntu 16.04.4 LTS, with Linux kernel 4.4.0-21. Each machine uses a 1Gbps Ethernet card connected to a dedicated local network. We use version 1.8 of the Intel SGX Linux SDK [24] and driver [23].

5.1 Server applications

One deployment model for SGX is to host network services on an untrusted cloud provider's hardware. We measure three widely-used Linux web servers, including **Lighttpd** [6] (v1.4.35), **Apache** [2] (v2.4.18), and **NGINX** [7] (v1.10). For each workload, we use ApacheBench [1] to download the web pages on a separate machine. The concurrency of ApacheBench is gradually increased during the experiment, to test the both the per-request latency and the overall throughput of the server. Figure 5 shows the throughput versus latency of these server applications in Graphene-SGX, Graphene and Linux. Each workload is discussed below.

Lighttpd [6] is a web server designed to be lightweight, yet robust enough for commercial uses. Lighttpd is multi-threaded; we test with 25 threads to process HTTP requests. By default, Lighttpd uses the `epoll_wait` system call to poll listening sockets. At peak throughput and load, both Graphene and Graphene-SGX have marginal overhead on either latency or throughput of the Lighttpd server. The overheads of



(a) Lighttpd (25 threads) **(b) Apache (5 processes)** **(c) NGINX (event-driven)**

Figure 5: Throughput versus latency of web server workloads, including Lighttpd, Apache, and NGINX, on native Linux, Graphene, and Graphene-SGX. We use an ApacheBench client to gradually increase load, and plot throughput versus latency at each point. Lower and further right is better.

Graphene are more apparent when the system is more lightly loaded, at 15–35% higher response time, or 13–26% lower throughput. Without SGX, Graphene induces 11–15% higher latency or 13-17% lower throughput over Linux; the remaining overheads are attributable to SGX—either hardware or our OS shield.

Apache [2] is one of the most popular production web servers. We test Apache using 5 preforked worker processes to service HTTP requests, in order to to evaluate the efficiency of Graphene-SGX across enclaves. This application uses IPC extensively—the preforked processes of a server use a System V semaphore to synchronize on each connection. Regardless of the workload, the response time on Graphene-SGX is 12–35% higher than Linux, due to the overhead of coordination across enclaves over encrypted RPC streams. The peak throughput achieved by Apache running in Graphene-SGX is 26% lower than running in Linux. In this workload, most of the overheads are SGX-specific, such as exiting enclaves when accessing the RPC, as non-SGX Graphene has only 2–8% overhead compared to Linux.

NGINX [7] is a relatively new web server designed for high programmability, for as a building block to implement different services. Unlike the other two web servers, NGINX is event-driven and mostly configured as single-threaded. Graphene-SGX currently only supports synchronous I/O at the enclave boundary, and so, under load, it cannot as effectively overlap I/O and computation as other systems that have batched and asynchronous system calls. Once sufficiently loaded, NGINX on both Graphene and Graphene-SGX performs worse than in a Linux process. The peak throughput of Graphene-SGX is 1.5× lower than Linux; without SGX, Graphene only reaches 79% of Linux's peak throughput. We expect that using tools like Eleos [43] to reduce exits would help this workload; in future work, we will improve asynchronous I/O in Graphene-SGX.

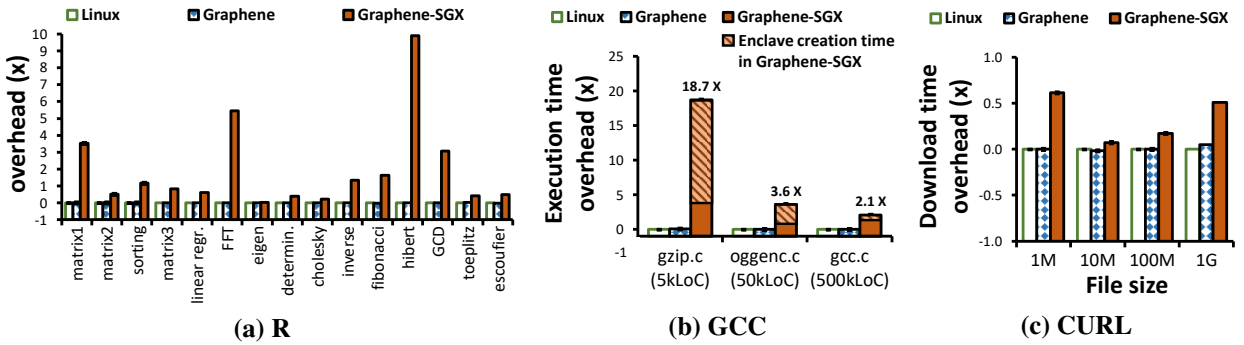
5.2 Command-Line Applications

We also evaluate the performance of a few commonly-used command-line applications. Three off-the-shelf applications are tested in our experiments: **R** (v3.2.3) for statistical computing [9]; **GCC** (v5.4), the general GNU C compiler [4]; **CURL** (v7.74), the default command-line web client on UNIX [3]. These applications are chosen because they are frequently used by Linux users, and each of them potentially be used in an enclave to handle sensitive data—either on a server or a client machine.

We evaluate the latency or execution time of these applications. In our experiments, both R and CURL have internal timing features to measure the wall time of individual operations or executions. On a Linux host, the time to start a library OS is higher than a simple process, but significantly lower than booting a guest OS in a VM or starting a container. Prior work measured Graphene (non-SGX) start time at 641 μs [52], whereas starting an empty Linux VM takes 10.3s and starting a Linux (LXC) container takes 200 ms [12].

On SGX, the enclave creation time is relatively higher, ranging from 0.5s (a 256MB enclave) to 5s (a 2G enclave), which is a fixed cost that any application framework will have to pay to run on SGX. Enclave creation time is determined by the latency of the hardware and the Intel kernel driver, and is primarily a function of the size of the enclave, which is specified at creation time because it affects the enclave signature. For non-server workloads that create multiple processes during execution, such as GCC in Figure 6, the enclave creation contributes a significant portion to the execution time overheads, illustrated as a stacked bar.

R [9] is a scripting language often used for data processing and statistical computation. With enclaves, users can process sensitive data on an OS they don't trust. We use an R benchmark suite developed by Urbanek et al. [8], which includes 15 CPU-bound workloads such as matrix computation and number processing. Graphene-SGX slows down by less than 100% on the majority of the workloads, excepts the ones which involve allocation and garbage collection: (`matrix1` creates and destroys matrices, and both `FFT` and `hilbert` involve heavy garbage collection.) Aside from garbage collection, these R benchmarks do not frequently interact with



(a) R (b) GCC (c) CURL

Figure 6: Performance overhead on desktop applications, including latency of R, execution time of GCC compilation, download time with CURL. The evaluation compares native Linux, Graphene, and Graphene-SGX.

the host. We further note that non-SGX Graphene is as efficient as Linux on all workloads, and these overheads appear to be SGX-specific. In our experience, garbage collection and memory management code in managed language runtime systems tends to be written with assumptions that do not match enclaves, such as a large, sparse address space or that memory can be demand paged nearly for free (SGX version 1 requires all memory to be mapped at creation); a useful area for future work would be to design garbage collection strategies that are optimized for enclaves.

GCC [4] is a widely-used C compiler. By supporting GCC in enclaves, developers can compile closed-source applications on customers' machines, without leaking the source code. GCC composes of multiple binaries, including `cc1` (compiler), `as` (assembler), and `ld` (linker). Therefore, GCC is a multi-process program using `execve`. We test the compilation of thee source files with varied sizes, using single C source files collected by MIT [5]. Each GCC execution typically creates five processes, and we run each process in a 256MB enclave by default. For a small workload like compiling `gzip.c` (5 kLoC), running in Graphene-SGX (4.1s) is 18.7× slower than Linux (0.2s). The bulk of this time is spent in enclave creation, taking 3.0s in total, while the whole execution inside the enclaves, including initialization of the library OS and OS shield, takes only 1.1s, or 4.2× overhead. For larger workloads like `oggenc.c` (50 kLoC) and `gcc.c` (500 kLoC), the overhead of Graphene-SGX is less significant. For `gcc.c` (500 kLoC), we have to enlarge one of the enclaves (`cc1`) to 2GB, but running on Graphene-SGX (53.1s) is only 2.1× slower than Linux (17.2s), and 7.1s is spent on enclave creation. The overhead of non-SGX Graphene on GCC is marginal.

CURL [3] is a command-line web downloader. Graphene-SGX can make CURL into a secure downloader that attests both server and client ends. We evaluate the total time to download a large file, ranging from 1MB to 1GB, from another machine running Apache. Graphene has marginal overhead on CURL, and Graphene-SGX adds 7–61% overhead to the downloading time of CURL, due to the latency of I/O.

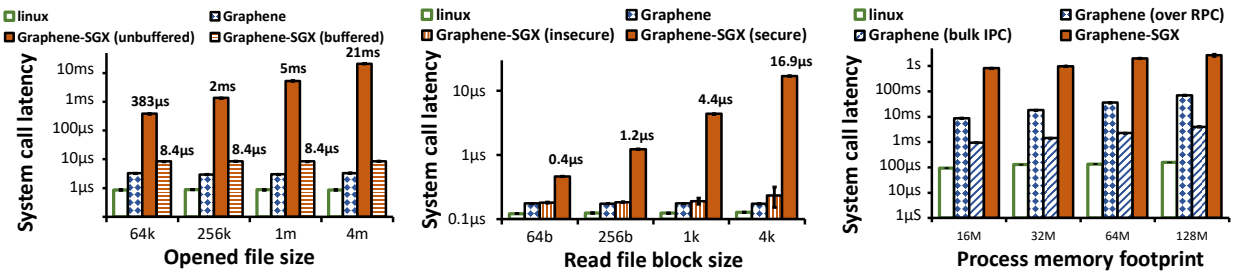
5.3 Performance Overhead Analysis

In this section we evaluate a few system operations that are heavily impacted by the Graphene-SGX design. We measure the `open`, `read`, and `fork` system calls using LMbench 2.5 [42]. A primary source of the overheads on these system calls is the cost of shielding applications, with run-time checks on the inputs. Cryptographic techniques are used to: (1) validate the file against the secure hash, at `open`, (2) check the file chunks against the Merkle tree, at `read`, and (3) establish a TLS connection over inter-enclave RPC, at `fork`. The remaining overheads contribute to exiting the enclave for host system calls, and bringing memory into the EPC (enclave page cache) or decrypting memory on a last-level cache miss.

Figure 7(a) shows the overhead for authenticating files in `open`. Depending on the file size, the latency of `open` on Graphene-SGX is 383μs (64KB file) to 21ms (4MB file), whereas on Linux, the latency is constant at 0.85μs. We note that this is where enclaves are at a disadvantage, as `open` normally does not need to read file content; whereas here Graphene-SGX uses `open` as a point at which to validate file content. For a subsequent `open`, when the Merkle tree is already generated, the overhead of simply exiting enclave for `open`, and searching the file list in the manifest, is about 9×.

One might be able to optimize further for cases where only part of a file is accessed with incremental hashing. However, in the common case where nearly all of the file is accessed, these costs are difficult to avoid when host file system is untrusted. Another opportunity is to create the Merkle tree offline, when the manifest is created.

Figure 7(b) shows the overhead for authenticating files in `read`, which is lower than `open`. Since the whole file has been verified at `open`, the sequential `read` only verifies the chunks of files it is reading from untrusted memory. Depending on the size of blocks being read, the latency on Graphene-SGX is 0.5μs (64-byte `read`)



(a) Open a file **(b) Read a file** **(c) Fork a process**

Figure 7: Latency of some expensive system calls in Graphene-SGX, including opening and reading a secured (authenticated) file, and forking a new process. The results are compared with native Linux and Graphene.

Components	**Graphene-SGX**	SCONE	Panoply
libc (ld, libm, pthread)	**1,292**	88	–
	(glibc-2.19)	(musl)	
Library OS	**34**	–	–
PAL / OS Shield	**22**	99	10
Total	**1,348**	187	10

Table 2: TCB size (in thousands of lines of code) of Graphene-SGX, SCONE, and Panoply.

to 16.9μs (4KB `read`). The latency of `read` on Linux is $\sim$0.1μs for any block size below 4KB. If the file is not authenticated, Graphene-SGX only copies the file contents into the buffer, and the overhead reduces to 48% (64-byte `read`) to 83% (4KB `read`).

Figure 7(c) shows the overhead of forking a process. As described in 4.3, the latency of `fork` in Graphene-SGX is affected by three factors: creation of a new enclave, local attestation of the integrity, and duplicating the process state over an encrypted RPC stream. Combining these factors, `fork` is one of the most expensive calls in Graphene-SGX. The default enclave size is 256MB. Our evaluation shows that the latency of forking a process is around 0.8s (16MB process) to 2.7s (128MB process), but can be more expensive if the parent process uses more memory. The trend matches the performance of Graphene without the bulk IPC optimization.

One way to further optimize `fork` is to reduce or avoid enclave creation time; one can potentially pre-launch a child enclave, and then migrate the process contents later when `fork` is called. There might be another opportunity to improve the latency of process migration, if copy-on-write sharing of enclave pages can be supported in future generations of SGX.

5.4 TCB Size and Shielded Functionality

In this section we measure the increase in TCB size of Graphene-SGX, as well as the OS functionality shielded by the framework. We compare to SCONE and Panoply, using numbers reported in their papers. A smaller TCB is generally easier to review or possibly verify, and is assumed to have fewer vulnerabilities.

Table 2 lists the lines of code in each components within the TCB of Graphene-SGX, SCONE, and Panoply. By comparing the total TCB size, Graphene-SGX is 9$\times$ larger than SCONE, and 134$\times$ larger than Panoply. However, the primary difference is the selection of libc: for maximum compatibility, Graphene uses glibc. SCONE uses the smaller musl libc, which lacks some features of glibc. Panoply excludes libc from its TCB, to fit into the range of automated formal verification, as they shield at the libc interface. In principle, Graphene could easily support musl as well as glibc for applications that do not need the additional features of glibc. We also see the benefit of removing unused code from libraries, especially in an unsafe language, similar to the approach taken in unikernels [38]. On balance, this choice of libc implementation is largely orthogonal to the issue of how general-purpose the shields are.

If we focus on the TCB size of the library OS and the shields, Graphene-SGX is 44% smaller than SCONE. We cannot analyze the size of SCONE because it is closed source. Panoply has a smaller TCB in its shield, but within the same order of magnitude. Panoply only shields 91 out of 256 supported POSIX functions; for context, POSIX 1003.1 defines 1,191 APIs [11].

All three of these compatibility layers or shields are within the same order of magnitude in code size, and the differences are likely correlated with different ranges of supported functionality. A recent study indicates that only order-of-magnitude differences in code size correlate with reported CVE vulnerabilities; within the same order-of-magnitude, the data is inconclusive that there is a meaningful difference in risk [25]. Thus, increased generality does not necessarily come with increased risk.

6 Related Work

Protection against untrusted OSes. Protecting applications from untrusted OSes predates hardware support. Virtual Ghost [20] uses both compile-time and run-time monitoring to protect an application from a potentially-compromised OS, but requires recompilation of the guest OS and application. Flicker [40], MUSHI [56],

SeCage [37], InkTag [21], and Sego [32] protect applications from untrusted OS using SMM mode or virtualization to enforce memory isolation between the OS and a trusted application. Koberl et al. [30], isolate software on low-cost embedded devices using a Memory Protection Unit. Li et al. [34] built a 2-way sandbox for x86 by separating the Native Client (NaCl) [55] sandbox into modules for sandboxing and service runtime to support application execution and use Trustvisor [39] to protect the piece of application logic from the untrusted OS. Jang et al. [26] build a secure channel to authenticate the application in the Untrusted area isolated by the ARM TrustZone technology. Song et al. [50] extend each memory unit with an additional tag to enforce fine-grained isolation at machine word granularity in the HDFI system.

Trusted execution hardware. XOM [35] is the first hardware design for trusted execution on an untrusted OS, with memory encryption and integrity protection similar to SGX. XOM supports containers of an application to be encrypted with a developer-chosen key. This encryption key is encrypted at design-time using a CPU-specific public key, and also used to tag cache lines that the containers are allowed to access. XOM realizes a similar trust model as SGX, except a few details, such as lack of paging support, and allowing `fork` by sharing the encryption key across containers.

Besides SGX, other hardware features have been introduced in recent years to enforce isolation for trusted execution. TrustZone [51] on ARM creates an isolated environment for trusted kernel components. Different from SGX, TrustZone separates the hardware between the trusted and untrusted worlds, and builds a trusted path from the trusted kernel to other on-chip peripherals. IBM SecureBlue++ [16] also isolates applications by encrypting the memory inside the CPU; SecureBlue++ is capable of nesting isolated environments, to isolate applications, guest OSes, hypervisors from each other.

AMD is introducing a feature in future chips called SEV (Secure Encrypted Virtualization) [27], which extends nested paging with encryption. SEV is designed to run the whole virtual machines, whereas SGX is designed for a piece of application code. SEV does not provide comparable integrity protection or the protection against replay attacks on SGX. Graphene-SGX provides the best of both worlds: unmodified applications with confidentiality and integrity protections in hardware.

Sanctum [19] is a RISC-V processor prototype that features a minimal and open design for enclaves. Sanctum also defends against some side channels, such as page fault address and cache timing, by virtualizing the page table and page fault handler inside each enclave.

SGX frameworks and applications. Besides shielding systems [14, 15, 49], SGX has been used in specific applications or to address other security issues. VC3 [45] runs MapReduce jobs in SGX enclaves. Similarly, Brenner et al. [17] run cluster services in ZooKeeper in an enclave, and transparently encrypt data in transit between enclaves. Ryoan [22] sandboxes a piece of untrusted code in the enclave to process secret data while preventing the loaded code from leaking secret data. Opaque [57] uses an SGX-protected layer on the Spark framework to generate oblivious relational operators that hide the access patterns of distributed queries. SGX has also been applied to securing network functionality [47], as well as inter-domain routing in Tor [29].

Several improvements to SGX frameworks have been recently developed, which can be integrated with applications on Graphene-SGX. Eleos [43] reduces the number of enclave exits by asynchronously servicing system calls outside of the enclaves, and enabling user-space memory paging. SGXBOUND [31] is a software technique for bounds-checking with low memory overheads, to fit within limited EPC size. T-SGX [48] combines SGX with Transactional Synchronization Extensions, to invoke a user-space handler for memory transactions aborted by page fault, to mitigate controlled-channel attacks. SGX-Shield [46] enables Address Space Layout Randomization (ASLR) in enclaves, with a scheme to maximize the entropy, and the ability to hide and enforce ASLR decisions. Glamdring [36] uses data-flow analysis at compile-time, to automatically determine the partition boundary in an application.

7 Conclusion

This paper demonstrates that the costs of running an unmodified application in SGX on a library OS are marginal compared to thinner shims. The major costs of using SGX are still hardware limitations of SGX. As SGX and similar technologies mature, these design choices may have more impact. In the interim, Graphene-SGX serves as a simple, open-source tool to quickly bring up existing applications on SGX, and then incrementally adapt the code to improve performance and security on SGX.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Mihai Christodorescu, for their insighful comments on the work. We also thank the users of Graphene for contributing bug reports, code patches, and suggestions for the project, as well as their patience with bug fixes. Part of this work was completed while Porter's primary affiliation was Stony Brook University. This work was supported in part by NSF grants CNS-1149229, CNS-1161541, CNS-1228839, CNS-1405641, VMware, and an SGX pre-release equipment loan from Intel.

References

[1] Apache HTTP benchmarking tool. http://httpd.apache.org/docs/2.4/programs/ab.html.

[2] Apache HTTP server project. https://httpd.apache.org/.

[3] CURL, command line tool and library for transferring data with url. https://curl.haxx.se.

[4] GCC, the GNU compiler collection. https://gcc.gnu.org.

[5] Large single compilation-unit C programs. http://people.csail.mit.edu/smcc/projects/single-file-programs/.

[6] Lighttpd. https://www.lighttpd.net/.

[7] NGINX. https://www.nginx.com/.

[8] R benchmark 2.5. http://www.math.tamu.edu/osg/R/R-benchmark-25.R.

[9] The R project for statical computing. https://www.r-project.org/.

[10] CVE-2009-2692. Available at MITRE, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2692, August 2009.

[11] IEEE International Standard for Information Technology – Portable Operating System Interface (POSIX) Base Specifications, Issue 7. Standard, IEEE, September 2009.

[12] K. Agarwal, B. Jain, and D. E. Porter. Containing the hype. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, APSys '15, 2015.

[13] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy at Proceedings of the ACM IEEE International Symposium on Computer Architecture (ISCA)*, 2013.

[14] 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 *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov 2016.

[15] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 267–283, 2014.

[16] R. Boivie and P. Williams. SecureBlue++: CPU support for secure executables. Technical report, IBM Research, 2013. Available at http://domino.research.ibm.com/library/cyberdig.nsf/papers/BE73A643EFE8274B85257B51006760C0.

[17] S. Brenner, C. Wulf, and R. Kapitza. Running ZooKeeper coordination services in untrusted clouds. In *10th Workshop on Hot Topics in System Dependability (HotDep 14)*, 2014.

[18] S. Checkoway and H. Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. *SIGPLAN Not.*, pages 253–264, March 2013.

[19] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security*, volume 16, pages 857–874, 2016.

[20] J. Criswell, N. Dautenhahn, and V. Adve. Virtual ghost: Protecting applications from hostile operating systems. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Citeseer, 2014.

[21] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. Inktag: secure applications on an untrusted operating system. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 265–278. ACM, 2013.

[22] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2016.

[23] Intel Corporation. Intel software guard extensions for Linux OS - Intel SGX driver. https://github.com/01org/linux-sgx.

[24] Intel Corporation. Intel software guard extensions for Linux OS - Intel SGX SDK. https://github.com/01org/linux-sgx.

[25] B. Jain, C.-C. Tsai, and D. E. Porter. A clairvoyant approach to evaluating software (in)security. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2017.

[26] J. S. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang. Secret: Secure channel between rich execution environment and trusted execution environment. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[27] D. Kaplan, J. Powell, and T. Woller. AMD memory encryption. White paper, April 2016. Available at `http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf`.

[28] S. Kim, J. Han, J. Ha, T. Kim, and D. Han. Enhancing security and privacy of tors ecosystem by using trusted execution environments. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, 2017.

[29] S. Kim, Y. Shin, J. Ha, T. Kim, and D. Han. A first step towards leveraging commodity trusted execution environments for network applications. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets)*, page 7. ACM, 2015.

[30] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, page 10. ACM, 2014.

[31] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer. SGXBOUNDS: Memory safety for shielded execution. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, EuroSys '17, 2017.

[32] Y. Kwon, A. M. Dunn, M. Z. Lee, O. S. Hofmann, Y. Xu, and E. Witchel. Sego: Pervasive trusted metadata for efficiently verified untrusted system services. *SIGOPS Oper. Syst. Rev.*

[33] Y. Kwon, A. M. Dunn, M. Z. Lee, O. S. Hofmann, Y. Xu, and E. Witchel. Sego: Pervasive trusted metadata for efficiently verified untrusted system services. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 277–290. ACM, 2016.

[34] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry. Minibox: A two-way sandbox for x86 native code. In *Proceedings of the USENIX Annual Technical Conference*, pages 409–420, 2014.

[35] D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. *ACM SIGOPS Operating Systems Review*, 37(5):178–192, 2003.

[36] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch. Glamdring: Automatic application partitioning for Intel SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, CA, 2017. USENIX Association.

[37] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2015.

[38] 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 ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[39] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, pages 143–158, 2010.

[40] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 315–328, 2008.

[41] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas. Intel software guard extensions (Intel SGX) support for dynamic memory management inside an enclave. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–9, New York, New York, USA, June 2016. ACM Press.

[42] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX Annual Technical Conference*, pages 23–23, 1996.

[43] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein. Eleos: ExitLess OS services for SGX enclaves. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2017.

[44] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 16–16, Berkeley, CA, USA, 2003. USENIX Association.

[45] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, pages 38–54. IEEE, 2015.

[46] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. SGX-Shield: Enabling address space layout randomization for SGX programs. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[47] M.-W. Shih, M. Kumar, T. Kim, and A. Gavrilovska. S-NFV: Securing NFV states by using SGX. In *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization (SDN-NFV Security)*, pages 45–48. ACM, 2016.

[48] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[49] S. Shinde, D. L. Tien, S. Tople, and P. Saxena. PANOPLY: Low-TCB Linux applications with SGX enclaves. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[50] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. HDFI: Hardware-assisted data-flow isolation. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 2016.

[51] ARM TrustZone technology overview. Available at `http://www.arm.com/products/processors/technologies/trustzone/index.php`.

[52] 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 ACM European Conference on Computer Systems (EuroSys)*, 2014.

[53] C.-C. Tsai, B. Jain, N. A. Abdul, and D. E. Porter. A study of modern Linux API usage and compatibility: What to support when you're supporting. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2016.

[54] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*. IEEE Institute of Electrical and Electronics Engineers, May 2015.

[55] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, pages 79–93. IEEE, 2009.

[56] N. Zhang, M. Li, W. Lou, and Y. T. Hou. Mushi: Toward multiple level security cloud with strong hardware level isolation. In *Military Communications Conference, 2012-MILCOM 2012*, pages 1–6. IEEE, 2012.

[57] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

PrivApprox: Privacy-Preserving Stream Analytics

https://PrivApprox.github.io

Do Le Quoc[†], Martin Beck[†], Pramod Bhatotia[*], Ruichuan Chen[‡], Christof Fetzer[†], Thorsten Strufe[†]

[†]TU Dresden [*]University of Edinburgh [‡]Nokia Bell Labs

Abstract

How to preserve users' privacy while supporting high-utility analytics for low-latency stream processing?

To answer this question: we describe the design, implementation and evaluation of PRIVAPPROX, a data analytics system for privacy-preserving stream processing. PRIVAPPROX provides three important properties: *(i)* Privacy: zero-knowledge privacy guarantee for users, a privacy bound tighter than the state-of-the-art differential privacy; *(ii)* Utility: an interface for data analysts to systematically explore the trade-offs between the output accuracy (with error estimation) and the query execution budget; *(iii)* Latency: near real-time stream processing based on a scalable "synchronization-free" distributed architecture.

The key idea behind our approach is to marry two techniques together, namely, *sampling* (used for approximate computation) and *randomized response* (used for privacy-preserving analytics). The resulting marriage is complementary — it achieves stronger privacy guarantees, and also improves the performance for stream analytics.

1 Introduction

Many online services continuously collect users' private data for real-time analytics. Much of this data arrives as a data stream and in huge volumes, requiring real-time stream processing based on distributed systems [1–3, 21].

In the current ecosystem of data analytics, the analysts usually have direct access to users' private data, and must be trusted not to abuse it. However, this trust has been violated in the past [28, 49, 62, 69]. A pragmatic ecosystem has two desirable, but contradictory design requirements: *(i)* stronger privacy guarantees for users, and *(ii)* high-utility stream analytics in real time. Users seek stronger privacy, while analysts strive for high-utility analytics in real time.

To meet these two design requirements, there is a surge of novel computing paradigms that address these concerns, albeit *separately*. Two such paradigms are *privacy-preserving analytics* to protect user privacy and *approximate computation* for real-time analytics.

Privacy-preserving analytics. Recent privacy-preserving analytics systems favor a distributed architecture to avoid central trust (see §8 for details), where users' private data is stored locally on their respective client devices. Data analysts use a publish-subscribe mechanism to run aggregate queries over the distributed private dataset of a large number of clients. Thereafter, such systems add noise to the aggregate output to provide useful privacy guarantees, such as differential privacy [32]. Unfortunately, these state-of-the-art systems normally deal with single-shot batch queries, and therefore, these systems cannot be used for real-time stream analytics.

Approximate computation. Approximate computation is based on the observation that many data analytics jobs are amenable to an approximate rather than the exact output (see §8 for details). Such applications include speech recognition, computer vision, machine learning, and recommender systems. For such an approximate workflow, it is possible to trade accuracy by computing over a subset (usually selected via a sampling mechanism) instead of the entire input dataset. Thereby, data analytics systems based on approximate computation can achieve low latency and efficient utilization of resources. However, the existing systems for approximate computation assume a centralized dataset, where the desired sampling mechanism can be employed. Thus, existing systems are not compatible with the distributed privacy-preserving analytics systems.

The marriage. In this paper, we make the observation that the two computing paradigms, i.e., privacy-preserving analytics and approximate computation, are complementary. Both paradigms strive for an approximate instead of the exact output, but they differ in their *means* and *goals* for approximation. Privacy-preserving analytics adds explicit *noise* to the aggregate query output to protect user privacy, whereas approximate computation relies on a representative *sampling* of the entire dataset to compute over only a subset of data items to enable low-latency/efficient analytics. Therefore, we marry these two existing paradigms together in order to leverage the benefits of both. The high-level

idea is to achieve privacy (via approximation) by directly computing over a subset of sampled data items (instead of computing over the entire dataset) and then adding an explicit noise for privacy preservation.

To realize this marriage, we designed an approximation mechanism that also achieves privacy-preserving goals for stream analytics. Our design (see Figure 1) targets a distributed setting, similar as aforementioned, where users' private data is stored locally on their respective personal devices, and an analyst issues a streaming query for analytics over the distributed private dataset of users. The analyst's streaming query is executed on the users' data periodically (a configurable epoch) and the query results are transmitted to a centralized aggregator via a set of proxies. The analyst interfaces with the aggregator to get the aggregate query output periodically.

We employ two core techniques to achieve our goal. Firstly, we employ *sampling* [60] directly at the user site for approximate computation, where each user randomly decides whether to participate in answering the query in the current epoch. Since we employ sampling at the data source, instead of sampling at a centralized infrastructure, we are able to squeeze out the desired data size (by controlling the sampling parameter) from the very first stage in the analytics pipeline, which is essential in low-latency environments.

Secondly, if the user participates in the query answering process, we employ a *randomized response* [37] mechanism to add noise to the query output at the user site, again locally at the source of the data in a decentralized fashion. In particular, each user locally randomizes the truthful answer to the query to achieve the differential privacy guarantees (§3.2.2). Since we employ noise addition at the source of data, instead of adding the explicit noise to the aggregate output at a trusted aggregator or proxies, we enable a truly "synchronization-free" distributed architecture, which requires *no coordination* among proxies and the aggregator for the mandated noise addition.

The last, but not the least, silver bullet of our design: it turns out that the combination of the two aforementioned techniques (i.e., sampling and randomized response) leads us to achieve zero-knowledge privacy [41], a privacy bound tighter than the state-of-the-art differential privacy [32].

To summarize, we present the design and implementation of a practical system for privacy-preserving stream analytics in real time. In particular, our system is a novel combination of the sampling and randomized response techniques, as well as a scalable "synchronization-free" routing scheme which employs a light-weight XOR-based encryption scheme [26]. The resulting system ensures zero-knowledge privacy, anonymization, and unlinkability for users (§2.2). Altogether, we make the following contributions:

- We present a marriage of the sampling and randomized response techniques to achieve improved performance and stronger privacy guarantees.

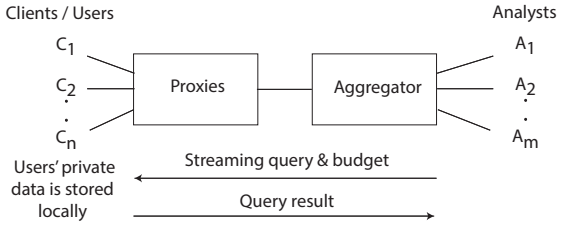


Figure 1: System overview.

- We present an adaptive query execution interface for analysts to systematically make a trade-off between the output accuracy and the query execution budget.
- We present a confidence metric on the output accuracy using a confidence interval to interpret the approximation due to sampling and randomization.

To empirically evaluate our approach, we implemented our design as a fully functional prototype in a system called PRIVAPPROX based on Apache Flink [21] and Apache Kafka [7]. In addition to stream analytics, we further extended our system to support privacy-preserving "historical" batch analytics over users' private datasets. The evaluation based on micro-benchmarks and real-world case studies shows that this marriage is, in fact, made in heaven!

2 Overview

2.1 System Architecture

PRIVAPPROX is designed for privacy-preserving stream analytics on distributed users' private dataset. Figure 1 depicts the high-level architecture of PRIVAPPROX. Our system consists of four main components: clients, proxies, aggregator, and analysts.

Clients locally store users' private data on their respective personal devices, and subscribe to queries from the system. *Analysts* publish streaming queries to the system, and also specify a query execution budget. The query execution budget can either be in the form of latency guarantees/SLAs, output quality/accuracy, or the computing resources for query processing. Our system ensures that the computation remains within the specified budget.

At a high-level, the system works as follows: a query published by an analyst is distributed to clients via the aggregator and proxies. Clients answer the analyst's query locally over the users' private data using a privacy-preserving mechanism. Client answers are transmitted to the aggregator via anonymizing *proxies*. The *aggregator* aggregates received answers from the clients to provide privacy-preserving stream analytics to the analyst.

2.2 System Model

Query model. PRIVAPPROX supports the SQL query language for *analysts* to formulate streaming queries, which are executed periodically at the clients as sliding window computations [14]. While queries can be complex, the results of a query are expressed as counts within histogram buckets, i.e.,

each bucket represents a range of the query's answer values. Specifically, each query answer is represented in the form of binary buckets, where each bucket stores a value '1' or '0' depending on whether or not the answer falls into the value range represented by that bucket. For example, an analyst can learn the driving speed distribution across all vehicles in San Francisco by formulating an SQL query "`SELECT speed FROM vehicle WHERE location='San Francisco'`". The analyst can then define 12 answer buckets on speed: '0', '1∼10', '11∼20', ···, '81∼90', '91∼100', and '>100'. If a vehicle is moving at 15 mph in San Francisco, it answers '1' for the third bucket and '0' for all others.

Our query model supports not only numeric queries as described above, but also non-numeric queries. For non-numeric queries, each bucket is specified by a matching rule or a regular expression. Note that, at first glance, our query model may appear simple; however, it has been shown to be effective for a wide-range of analytics algorithms [19, 20].

Threat model. *Analysts* are potentially malicious. They may try to violate the PRIVAPPROX's privacy model (described later), i.e., de-anonymize clients, build profiles through the linkage of queries and answers, or remove the added noise from answers.

Clients are potentially malicious. They could generate false or invalid responses to distort the query result for the analyst. However, we do not defend against the Sybil attack [31], which is beyond the scope of this work [75].

Proxies are also potentially malicious. They may transmit messages between clients and the aggregator in contravention of our system protocols. PRIVAPPROX includes at least two proxies, and there are at least two proxies which do not collude with each other.

The *aggregator* is assumed to be honest-but-curious. The aggregator faithfully conforms to the system protocols, but may try to exploit the information about clients. The aggregator does not collude with any proxy nor the analyst.

Finally, we assume that all the end-to-end communications use authenticated and confidential connections (e.g., protected by long-lived TLS connections), and no system component could monitor all network traffic.

Privacy model. Our privacy properties include: *(i)* zero-knowledge privacy, *(ii)* anonymity, and *(iii)* unlinkability.

All aggregate query results in the system are independently produced under the *zero-knowledge privacy* guarantees [41]. The zero-knowledge privacy metric builds upon differential privacy [32], and provides a tighter bound on privacy guarantees compared to differential privacy. Informally, zero-knowledge privacy states that essentially everything that an adversary can learn from the output of an zero-knowledge private mechanism could also be learned using the aggregate information. *Anonymity* means that no system component can associate query answers or query requests with a specific client. Finally, *unlinkability* means that no system component can join any pair of query requests or answers to the same client, even to the same anonymous client.

We give a sketch of the privacy analysis in §4, while we also provide the formal definition, analysis, and proof in the technical report [64].

3 Design

PRIVAPPROX consists of two main phases (see Figure 1): *submitting queries* and *answering queries*. In the first phase, an analyst submits a query (along with the execution budget) to clients via the aggregator and proxies. In the second phase, the query is answered by the clients in the reverse direction.

3.1 Submitting Queries

To perform statistical analysis over users' private data streams, an analyst creates a query using the query model described in §2.2. In particular, each query consists of the following fields, and is signed by the analyst for non-repudiation:

$$Query := \langle Q_{ID}, SQL, A[n], f, w, \delta \rangle \quad (1)$$

- Q_{ID} denotes a unique identifier of the query. This can be generated by concatenating the identifier of the analyst with a serial number unique to the analyst.
- *SQL* denotes the actual SQL query, which is passed on to clients and executed on their respective personal data.
- $A[n]$ denotes the format of a client's answer to the query. The answer is an n-bit vector where each bit associates with a possible answer value in the form of a "0" or "1" per index (or answer value range).
- f denotes the answer frequency, i.e., how often the query needs to be executed at clients.
- w denotes the window length for sliding window computations [13]. For example, an analyst may only want to aggregate query results for the last ten minutes, which means the window length is ten minutes.
- δ denotes the sliding interval for sliding window computations. For example, an analyst may want to update the query results every one minute, and so the sliding interval is set to one minute.

After forming the query, the analyst sends the query, along with the query execution budget, to the aggregator. Once receiving the pair of the query and query budget from the analyst, the aggregator first converts the query budget into system parameters for sampling (s) and randomization (p,q). We explain these system parameters in the next section §3.2. Hereafter, the aggregator forwards the query and the converted system parameters to clients via proxies.

3.2 Answering Queries

After receiving the query and system parameters, we next explain how the query is answered by clients and processed by the system to produce the result for the analyst. The query answering process involves four steps including *(i)* sampling at clients for low-latency approximation; *(ii)* randomizing answers for privacy preservation; *(iii)* transmitting answers

via proxies for anonymization and unlinkability; and finally, *(iv)* aggregating answers with error estimation to give a confidence level on the approximate result.

3.2.1 Step I: Sampling at Clients

We make use of approximate computation to achieve low-latency execution by computing over a subset of data items instead of the entire input dataset. Specifically, our work builds on sampling-based techniques [8, 9, 42, 53, 65] in the context of "Big Data" analytics. Since we aim to keep the private data stored at individual clients, PRIVAPPROX applies an input data sampling mechanism locally at the clients. In particular, we use *Simple Random Sampling* (SRS) [60].

Simple Random Sampling (SRS). SRS is considered as a fair way of selecting a sample from a given population since each individual in the population has the same chance of being included in the sample. We make use of SRS at the clients to select clients that will participate in the query answering process. In particular, the aggregator passes the *sampling parameter* (s) on to clients as the probability of participating in the query answering process. Thereafter, each client flips a coin with the probability based on the sampling parameter (s), and decides whether to participate in answering a query. Suppose that we have a population of U clients, and each client i has an answer a_i. We want to calculate the sum of these answers across the population, i.e., $\sum_{i=1}^{U} a_i$. To compute an approximate sum, we apply the SRS at clients to get a sample of U' clients. The estimated sum is then calculated as follows:

$$\hat{\tau} = \frac{U}{U'} \sum_{i=1}^{U'} a_i \pm error \tag{2}$$

Where the error bound *error* is defined as:

$$error = t\sqrt{\widehat{Var}(\hat{\tau})} \tag{3}$$

Here, t is a value of the t-distribution with $U'-1$ degrees of freedom at the $1-\alpha/2$ level of significance, and the estimated variance $\widehat{Var}(\hat{\tau})$ of the sum is:

$$\widehat{Var}(\hat{\tau}) = \frac{U^2}{U'}\sigma^2\left(\frac{U-U'}{U}\right) \tag{4}$$

Where σ^2 is the sample variance of the sum.

Note that, in this paper, we assume that all clients produce the input stream with data items following the same distribution, i.e., all clients' data streams belong to the same stratum. We further extend our sampling mechanism with the *stratified sampling* technique [53] to deal with varying distributions of data streams. We cover the algorithm and evaluation of stratified sampling in the technical report [64].

3.2.2 Step II: Answering Queries at Clients

Clients that participate in the query answering process make use of the *randomized response* technique [37] to preserve answer privacy, with *no* synchronization among clients.

Randomized response. Randomized response protects user's privacy by allowing individuals to answer sensitive queries without providing truthful answers all the time, yet it allows analysts to collect statistical results. Randomized response works as follows: suppose an analyst sends a query to individuals to obtain the statistical result about a sensitive property. To answer the query, a client locally randomizes its answer to the query [37]. Specifically, the client flips a coin, if it comes up heads, then the client responds its truthful answer; otherwise, the client flips a second coin and responds "Yes" if it comes up heads or "No" if it comes up tails. The privacy is preserved via the ability to refuse responding truthful answers.

Suppose that the probabilities of the first coin and the second coin coming up heads are p and q, respectively. The analyst receives N randomized answers from individuals, among which R_y answers are "Yes". Then, the number of original truthful "Yes" answers before the randomization process can be estimated as:

$$E_y = \frac{R_y - (1-p)\times q \times N}{p} \tag{5}$$

Suppose A_y and E_y are the actual and the estimated numbers of the original truthful "Yes" answers, respectively. The accuracy loss η is then defined as:

$$\eta = \left|\frac{A_y - E_y}{A_y}\right| \tag{6}$$

It has been proven in [36] that, the randomized response mechanism achieves ε-differential privacy [32], where:

$$\varepsilon = \ln\left(\frac{\Pr[Response=Yes|Truth=Yes]}{\Pr[Response=Yes|Truth=No]}\right) \tag{7}$$

More specifically, the above randomized response mechanism achieves ε-differential privacy, where:

$$\varepsilon = \ln\left(\frac{p+(1-p)\times q}{(1-p)\times q}\right) \tag{8}$$

The reason is that, if a truthful answer is "Yes", then with the probability of '$p+(1-p)\times q$', the randomized answer will still remain "Yes". Otherwise, if a truthful answer is "No", then with the probability of '$(1-p)\times q$', the randomized answer will become "Yes".

It is worth mentioning that, combining the randomized response with the sampling technique described in Step I, we achieve not only differential privacy but also zero-knowledge privacy [41] which is a privacy bound tighter than differential privacy. We sketch out the proof in §4, with details in the technical report [64].

3.2.3 Step III: Transmitting Answers via Proxies

After producing randomized responses, clients transmit them to the aggregator via the proxies. To achieve anonymity

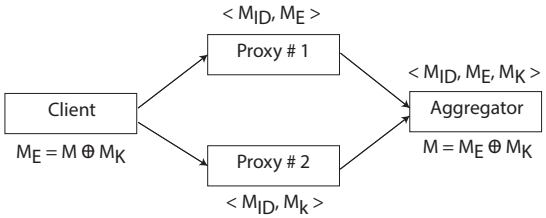


Figure 2: XOR-based encryption with two proxies.

and unlinkability of the clients against the aggregator and analysts, we utilize the XOR-based encryption together with source rewriting, which has been used for anonymous communications [26, 27, 30, 67].

XOR-based encryption. At a high-level, the XOR-based encryption employs extremely efficient bit-wise XOR operations as its cryptographic primitive compared to expensive public-key cryptography. This allows us to support resource-constrained clients, e.g., smartphones and sensors. The underlying idea of this encryption is simple: if Alice wants to send a message M of length l to Bob, then Alice and Bob share a secret M_K (in the form of a random bit-string of length l). To transmit the message M privately, Alice sends an encrypted message '$M_E = M \oplus M_K$' to Bob, where '$\oplus$' denotes the bit-wise XOR operation. To decrypt the message, Bob again uses the bit-wise XOR operation: $M = M_E \oplus M_K$.

Specifically, we apply the XOR-based encryption to transmit clients' randomized answers as follows. At first, each randomized answer is concatenated with its associated query identifier Q_{ID} to build a message M:

$$M = Q_{ID}, RandomizedAnswer \tag{9}$$

Thereafter, the client generates $(n-1)$ random l-bit key strings M_{K_i} with $2 \leq i \leq n$ using a cryptographic pseudo-random number generator (PRNG) seeded with a cryptographically strong random number. The XOR of all $(n-1)$ key strings together forms the secret M_K.

$$M_K = \bigoplus_{i=2}^{n} M_{K_i} \tag{10}$$

Next, the client performs an XOR operation with M and M_K to produce an encrypted message M_E.

$$M_E = M \oplus M_K \tag{11}$$

As a result, the message M is split into n messages $\langle M_E, M_{K_2}, \cdots, M_{K_n} \rangle$. Afterwards, a unique message identifier M_{ID} is generated, and sent along with the split messages to the n proxies via anonymous channels enabled by source rewriting [30, 67].

$$\begin{aligned} &\text{Client} \longrightarrow \text{Proxy1} : \langle M_{ID}, M_E \rangle \\ &\text{Client} \longrightarrow \text{Proxy}i : \langle M_{ID}, M_{K_i} \rangle \end{aligned} \tag{12}$$

Upon receiving the messages (either $\langle M_{ID}, M_E \rangle$ or $\langle M_{ID}, M_{K_i} \rangle$) from clients, the n proxies transmit these messages to the aggregator.

The message identifier M_{ID} ensures that M_E and all associated M_{K_i} will be joined later to decrypt the original message M at the aggregator. Note that, $\langle M_{ID}, M_E \rangle$ and all $\langle M_{ID}, M_{K_i} \rangle$ are computationally indistinguishable, which hides from the proxies if the received data contains the encrypted answer or is just a pseudo-random bit string.

3.2.4 Step IV: Generating Result at the Aggregator

At the aggregator, all data streams ($\langle M_{ID}, M_E \rangle$ and $\langle M_{ID}, M_{K_i} \rangle$) are received, and can be joined together to obtain a unified data stream. Specifically, the associated M_E and M_{K_i} are paired by using the message identifier M_{ID}. To decrypt the original randomized message M from the client, the XOR operation is performed over M_E and M_K: $M = M_E \oplus M_K$ with M_K being the XOR of all M_{K_i}: $M_K = \bigoplus_{i=2}^{n} M_{K_i}$. As the aggregator cannot identify which of the received messages is M_E, it just XORs all the n received messages to decrypt M.

The joined answer stream is processed to produce the query results as a sliding window. For each window, the aggregator first adapts the computation window to the current start time t by removing all old data items, with $timestamp < t$, from the window. Next, the aggregator adds the newly incoming data items into the window. Then, the answers in the window are decoded and aggregated to produce the query results for the analyst. Each query result is an estimated result which is bound to a range of error due to the approximation. The aggregator estimates this error bound using equation 3 and produces a confidence interval for the result as: $queryResult \pm errorBound$. The entire process is repeated for every window.

Note that an adversarial client might answer a query many times in an attempt to distort the query result. However, we can handle this problem, for example, by applying the *triple splitting* technique [26].

Error bound estimation. We provide an error bound estimation for the aggregate query results. The accuracy loss in PRIVAPPROX is caused by two processes: *(i)* sampling and *(ii)* randomized response. Since the accuracy loss of these two processes is statistically independent (see §6), we estimate the accuracy loss of each process separately. Furthermore, Equation 2 indicates that the error induced by sampling can be described as an additive component of the estimated sum. The error induced by randomized response is contained in the a_i values in Equation 2. Therefore, independent of the error induced by randomized response, the error coming from sampling is simply being added upon. Following this, we sum up both independently estimated errors to provide the total error bound of the query results.

To estimate the accuracy loss of the randomized response process, we make use of an experimental method. We

run several micro-benchmarks at the beginning of the query answering process (without performing the sampling process) to estimate the accuracy loss caused by randomized response. We measure the accuracy loss using Equation 6.

On the other hand, to estimate the accuracy loss of the sampling process, we apply the statistical theory of the sampling techniques. In particular, we first identify a desired confidence level, e.g., 95%. Then, we compute the margin of error using Equation 3. Note that, to use this equation the sampling distribution must be nearly normal. According to the Central Limit Theorem (CLT), when the sample size U' is large enough (e.g., ≥ 30), the sampling distribution of a statistic becomes close to the normal distribution, regardless of the underlying distribution of values in the dataset [72].

3.3 Practical Considerations

We next present two design enhancements to further improve the practicality of PRIVAPPROX.

3.3.1 Historical Analytics

In addition to providing real-time data analytics, we further extend PRIVAPPROX to support historical analytics. The historical analytics workflow is essential for the data warehousing setting, where analysts wish to analyze user behaviors over a longer time period. To facilitate historical analytics, we support the batch analytics over user data at the aggregator. The analyst can analyze users' responses stored in a fault-tolerant distributed storage (e.g., HDFS) at the aggregator to get the aggregate query result over the desired time period.

We also extend the adaptive execution interface for historical analytics, where the analyst can specify query execution budget, for example, to suit dynamic pricing in spot markets in the cloud deployment. Based on the query budget, we can perform an additional round of sampling at the aggregator to ensure that the batch analytics computation remains within the query budget (see the evaluation details in the technical report [64]).

3.3.2 Query Inversion

In the current setting, some queries may result in very few truthful "Yes" answers in users' responses. For such cases, PRIVAPPROX can only achieve lower utility of query results because the fraction of truthful "Yes" answers is distant from the second randomization parameter q (see experimental results in §6). For instance, if q is set to a high value (e.g., $q=0.9$), having only a few truthful "Yes" answers will affect the overall utility of the query result. To address this issue, we propose a *query inversion* mechanism. If the fraction of truthful "Yes" answers is too small or too large compared to the q value, then the analysts can invert the query to calculate the truthful "No" answers instead of the truthful "Yes" answers. In this way, the fraction of truthful "No" answers gets closer to q, resulting in a higher utility of the query result.

4 Privacy Analysis

PRIVAPPROX achieves the strong privacy properties *(i)* differential privacy and *(ii)* zero-knowledge privacy as introduced in §2.2. This section only provides a sketch of the full proof. The detailed proof along with the empirical evaluation is available in the technical report [64].

The basic idea is that all data from the clients is already differentially private due to the use of randomized response. Furthermore, the combination with sampling at the clients makes it zero-knowledge private as well. Following the privacy definitions [32, 41], any computation upon the results of differentially as well as zero-knowledge private algorithms is guaranteed to be private.

Intuitively, differential privacy limits the information that can be learned about any individual i by the difference occurring from either including i's sensitive data in a differentially private computation or not. Zero-knowledge privacy on the other hand also gives the adversary access to aggregate information about the remaining individuals. Essentially, everything that can be learned about individual i can also be learned by having access to some aggregate information upon them.

***(i)* Differential privacy.** Differential privacy is already fulfilled by randomized response [36]. However, due to the use of client-side sampling, a tighter privacy bound can be derived. Consequently, we show that sampling and randomize response commute and how to derive the combined bound given the sampling and randomize response parameters. The commutative property is shown by showing statistical indistinguishability of applying a sampling and randomize response in that order and vice versa. Furthermore, we show that sampling can be decomposed into pre- and post-sampling by leveraging the commutative property of multiplication.

***(ii)* Zero-knowledge privacy.** The zero-knowledge privacy property follows from combining a differentially private algorithm (randomized response) with an aggregation function (sampling) as given in the seminal work on zero-knowledge privacy [40]. More detail is available in the technical report [64].

5 Implementation

We implemented PRIVAPPROX as an end-to-end stream analytics system. Figure 3 presents the architecture of our prototype. Our system implementation consists of three main components: *(i)* clients, *(ii)* proxies, and *(iii)* the aggregator.

First, the query and the execution budget specified by the analyst are processed by the `initializer` module to decide on the sampling parameter (s) and the randomization parameters (p and q). These parameters along with the query are then sent to the clients.

Clients. We implemented Java-based clients for mobile devices as well as for personal computers. A client makes use of the sampling parameter (based on the `sampling` module) to decide whether to participate in the query answering process (§3.2.1). If the client decides to participate then

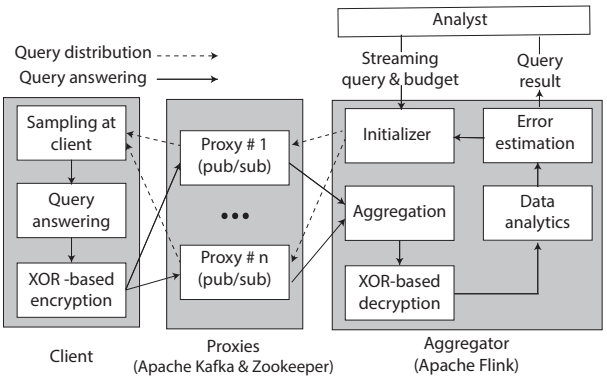


Figure 3: PRIVAPPROX architecture. Historical analytics pipeline at the aggregator is not shown for clarity.

the `query answer` module is used to execute the input query on the local user's private data stored in `SQLite` [5]. The client makes use of the randomized response to execute the query (§3.2.2). Finally, the randomized answer is encrypted using the `XOR-based encryption` module; thereafter, the encrypted message and the key messages are sent to the aggregator via proxies (§3.2.3).

Proxies. We implemented proxies based on Apache Kafka (which internally uses Apache Zookeeper [4] for fault tolerance). In Kafka, a *topic* is used to define a stream of data items. A stream *producer* can publish data items to a topic, and these data items are stored in Kafka servers called *brokers*. Thereafter, a *consumer* can subscribe to the topic and consume the data items by pulling them from the brokers. In particular, we make use of Kafka APIs to create two main topics: *key* and *answer* for transmitting the key message stream and the encrypted answer stream in the XOR-based encryption protocol, respectively (§3.2.3).

Aggregator. We implemented the aggregator using Apache Flink for real-time stream analytics and also for historical batch analytics. At the aggregator, we first make use of the join method (using the `aggregation` module) to combine the two data streams: *(i)* encrypted answer stream and *(ii)* key stream. Thereafter, the combined message stream is decoded (using the `XOR-based decryption` module) to reproduce the randomized query answers. These answers are then forwarded to the analytics module. The `analytics` module processes the answers to provide the query result to the analyst. Moreover, the `error estimation module` is used to estimate the error (§3.2.4), which we implemented using the Apache Common Math library. If the error exceeds the error bound target, a feedback mechanism is activated to re-tune the sampling and randomization parameters to provide higher utility in the subsequent epochs.

6 Evaluation: Microbenchmarks

We first evaluate PRIVAPPROX using microbenchmarks.

#I: Effect of sampling and randomization parameters. We measure the effect of randomization parameters on the utility and the privacy guarantee of the query results. In particular, the utility is measured by the query results' accuracy loss (Equation 6), and privacy is measured by the level of achieved zero-knowledge privacy (Equation 19 in the technical report [64]). In the experiment, we randomly generated 10,000 original answers, 60% of which are "Yes" answers. The sampling parameter s is set to 0.6.

Table 1: Utility and privacy of query results with different randomization parameters p and q.

p	q	Accuracy loss (η)	Privacy Level (ε)
0.3	0.3	0.0278	1.7047
	0.6	0.0262	1.3862
	0.9	0.0268	1.2527
0.6	0.3	0.0141	2.5649
	0.6	0.0128	2.0476
	0.9	0.0136	1.7917
0.9	0.3	0.0098	4.1820
	0.6	0.0079	3.5263
	0.9	0.0102	3.1570

Table 1 shows that different settings of the two randomization parameters, p and q, do affect the utility and the privacy guarantee of the query results. The higher p means the higher probability that a client responds with its truthful answer. As expected, this leads to higher utility (i.e., smaller accuracy loss η) but weaker privacy guarantee (i.e., higher privacy level ε). In addition, Table 1 also shows that the closer we set the probability q to the fraction of truthful "Yes" answers (i.e., 60% in this microbenchmark), the higher utility the query result provides. Nevertheless, to meet the utility and privacy requirements in various scenarios, we should carefully choose the appropriate p and q. In practice, the selection of the ε value depends on real-world applications [54].

We also measured the effect of sampling parameter on the accuracy loss. Figure 4 (a) shows that the accuracy loss decreases with the increase of sampling fraction, regardless of the settings of randomization parameters p and q. The benefits reach diminishing returns after the sampling fraction of 80%. The system operator can set the sampling fraction using resource prediction model [78–80] for any given SLA.

#II: Error estimation. To analyze the accuracy loss, we first measured the accuracy loss caused by sampling and randomized response *separately*. For comparison, we also computed the total accuracy loss after running the two processes in succession as in PRIVAPPROX. In this experiment, we set the number of original answers to 10,000 with 60% of which being "Yes" answers. We measured the accuracy loss of the randomized response process by setting the sampling parameter to 100% ($s=1$) and the randomization parameters p and q to 0.3 and 0.6, respectively. Meanwhile, we measured the accuracy loss of the sampling process without the randomized response process by setting p to 1.

Figure 4 (b) indicates that the accuracy loss caused by the two processes is statistically independent of each other. In addition, the accuracy loss of the two processes can effectively be added together to calculate the total accuracy loss.

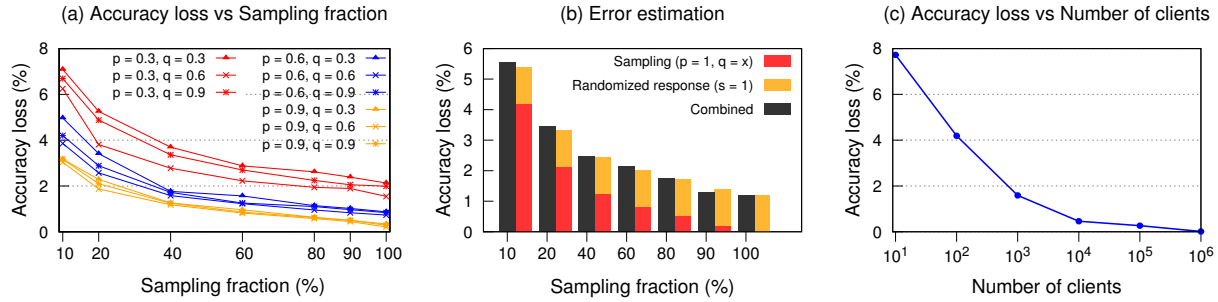


Figure 4: (a) Accuracy loss with varying sampling and randomization parameters. **(b)** Accuracy loss caused by sampling and randomized response processes, combined and individually. **(c)** Accuracy loss with varying numbers of clients.

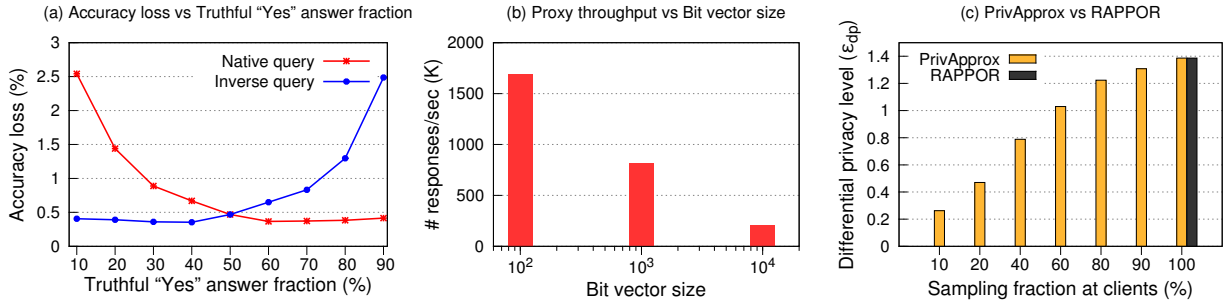


Figure 5: (a) Accuracy loss of the native and inversed query results with different fractions of truthful "Yes" answers. **(b)** Throughput at proxies with different sizes of the query answer. **(c)** Comparison between PRIVAPPROX and RAPPOR.

Table 2: Comparison of crypto overheads (# operations/sec). The public-key crypto schemes use a 1024-bit key.

	Encryption						Decryption					
	Phone		Laptop		Server		Phone		Laptop		Server	
RSA [10]	937	16×	2,770	341×	4,909	275×	126	25890×	698	23666×	859	26401×
Goldwasser [27]	2,106	7×	17,064	55×	22,902	59×	127	25686×	6,329	2610×	7,068	3209×
Paillier [66]	116	129×	489	1930×	579	2335×	72	45308×	250	66076×	309	73392×
PRIVAPPROX	15,026		943,902		1,351,937		3,262,186		16,519,076		22,678,285	

#III: Effect of the number of clients. We next analyzed how the number of participating clients affects the utility of the results. In this experiment, we fix the sampling and randomization parameters s, p and q to 0.9, 0.9 and 0.6, respectively, and set the fraction of truthful "Yes" answers to 60%.

Figure 4 (c) shows that the utility of query results improves with the increase of the number of participating clients, and few clients (e.g., <100) may lead to low-utility query results.

Note that increasing the number of participating clients leads to higher network overheads. However, we can tune the number of clients using the sampling parameter s and thus decrease the network overhead (see §7.2 #II).

#IV: Effect of the fraction of truthful answers. We measured the utility of both the native and the inversed query results with different fractions of truthful "Yes" answers. In this experiment, we still keep the sampling and randomization parameters s, p and q to 0.9, 0.9 and 0.6, respectively, and set the total number of answers to 10,000.

Figure 5 (a) shows that PRIVAPPROX achieves higher utility as the fraction of truthful "Yes" answers gets closer to 60% (i.e., the q value). In addition, when the fraction of truthful "Yes" answers y is too small compared to the q value (e.g., $y=0.1$), the accuracy loss is quite high at 2.54%. However, by using the query inversion mechanism (§3.3.2), we can significantly reduce the accuracy loss to 0.4%.

#V: Effect of answer's bit-vector sizes. We measured the throughput at proxies with various bit-vector sizes of client answers (i.e., $A[n]$ in §3.1). We conducted this experiment with a 3-node cluster (see §7.1 for the experimental setup). Figure 5 (b) shows that the throughput, as expected, is inversely proportional to the answer's bit-vector sizes.

#VI: Computational overhead of crypto operations. We compared the computational overhead of crypto operations used in PRIVAPPROX and prior systems. In particular, these crypto operations are XOR in PRIVAPPROX, RSA in [10], Goldwasser-Micali in [27], and Paillier in [66]. In this experiment, we measured the number of crypto operations that can be executed on: *(i)* Android Galaxy mini III smartphone

Table 3: Throughput (# operations/sec) at clients.

# operations/sec	Phone	Laptop	Server
SQLite read	1,162	19,646	23,418
Randomized response	168,938	418,668	1,809,662
XOR encryption	15,026	943,902	1,351,937
Total	1,116	17,236	22,026

running Android 4.1.2 with a 1.5 GHz CPU; *(ii)* MacBook Air laptop with a 2.2 GHz Intel Core i7 CPU running OS X Yosemite 10.10.2; and *(iii)* Linux server running Linux 3.15.0 equipped with a 2.2 GHz CPU with 32 cores.

Table 2 shows that the XOR operation is extremely efficient compared with the other crypto mechanisms. This highlights the importance of XOR encryption in our design.

#VII: Throughput at clients. We measured the throughput at clients. In particular, we measured the number of operations per second that can be executed at clients for the query answering process. In this experiment, we used the same set of devices as in the previous experiment. Table 3 presents the throughput at clients. To closely investigate the overheads, we measured the individual throughput of three sub-processes in the query answering process: *(i)* database read, *(ii)* randomized response, and *(iii)* XOR encryption. The result indicates that the performance bottleneck in the answering process is actually the database read operation.

#VIII: Comparison with related work. First, we compared PRIVAPPROX with SplitX [26], a high-performance privacy-preserving analytics system. Since PRIVAPPROX and SplitX share the same architecture, we compare the latency incurred at proxies in both systems.

Figure 6 shows that, with different numbers of clients, the latency incurred at proxies in PRIVAPPROX is always nearly one order of magnitude lower than that in SplitX. The reason is simple: unlike PRIVAPPROX, SplitX requires synchronization among its proxies to process query answers in a privacy-preserving fashion. This synchronization creates a significant delay in processing query answers, making SplitX unsuitable for dealing with large-scale stream analytics. More specifically, in SplitX, the processing at proxies consists of a few sub-processes including adding noise to answers, answer transmission, answer intersection, and answer shuffling; whereas, in PRIVAPPROX, the processing at proxies contains only the answer transmission. Figure 6 also shows that with 10^6 clients, the latency at SplitX is 40.27 sec, whereas PRIVAPPROX achieves a latency of just 6.21 sec, resulting in a 6.48× speedup compared with SplitX.

Next, we compared PRIVAPPROX with a recent privacy-preserving analytics system called RAPPOR [73]. Similar to PRIVAPPROX, RAPPOR applies a randomized response mechanism to achieve differential privacy. However, RAPPOR is not designed for stream analytics, and therefore, we compared PRIVAPPROX with RAPPOR for privacy only. To make an "apples-to-apples" comparison between PRIVAPPROX and RAPPOR in terms of privacy, we make a mapping

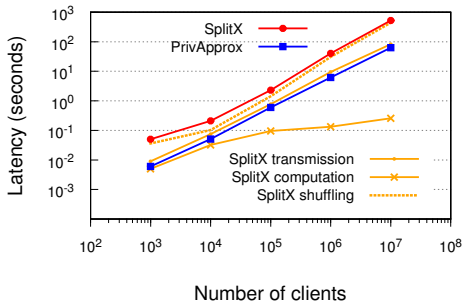


Figure 6: Comparison between SplitX and PRIVAPPROX.

between the system parameters of the two systems. We set the sampling parameter $s=1$, and the randomized parameters $p=1-f$, $q=0.5$ in PRIVAPPROX, where f is the parameter used in the randomized response process of RAPPOR [73]. In addition, we set the number of hash functions used in RAPPOR to 1 ($h=1$) for a fair comparison. In doing so, the two systems have the same randomized response process. However, since PRIVAPPROX makes use of the sampling mechanism before performing the randomized response, PRIVAPPROX achieves stronger privacy. Figure 5 (c) shows the differential privacy level of RAPPOR and PRIVAPPROX with different sampling fractions s. It is worth mentioning that, by applying the sampling mechanism, PRIVAPPROX achieves stronger privacy (i.e., zero-knowledge privacy) for clients.

7 Evaluation: Case Studies

We next present our experience of using PRIVAPPROX in the following two case studies: *(i)* New York City (NYC) taxi ride, and *(ii)* household electricity consumption.

7.1 Experimental Setup

Cluster setup. We used a cluster of 44 nodes connected via a Gigabit Ethernet. Each node contains 2 Intel Xeon quad-core CPUs and 8 GB of RAM running Debian 5.0. We deployed two proxies with Apache Kafka, each of which consists of 4 Kafka broker nodes and 3 Zookeeper nodes. We used 20 nodes to deploy Apache Flink as the aggregator. In addition, we employed the remaining 10 nodes to replay the datasets to generate data streams for the evaluation.

Datasets. For the first case study, we used the *NYC Taxi Ride* dataset from the DEBS 2015 Grand Challenge [51]. For the second case study, we used the *Household Electricity Consumption* dataset [6].

Queries. For the NYC taxi ride case study, we created a query: "*What is the distance distribution of taxi rides in New York?*". We defined the query answer with 11 buckets as follows: [0, 1) mile, [1, 2) miles, [2, 3) miles, [3, 4) miles, [4, 5) miles, [5, 6) miles, [6, 7) miles, [7, 8) miles, [8, 9) miles, [9, 10) miles, and [10, $+\infty$) miles.

For the second case study, we defined a query to analyze the electricity usage distribution of households over the past 30 minutes. The query answer format is as follows: [0, 0.5]

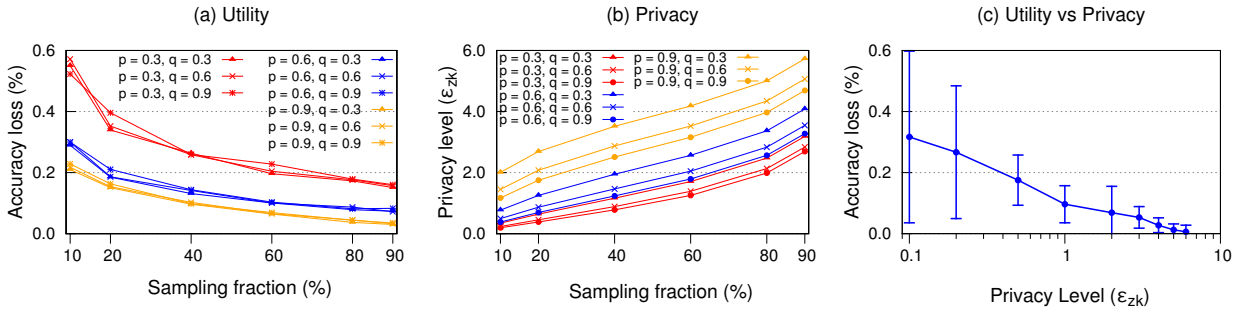


Figure 7: Results from the NYC Taxi case study with varying sampling and randomization parameters: **(a)** Utility, **(b)** Privacy level, **(c)** Comparison between utility and privacy.

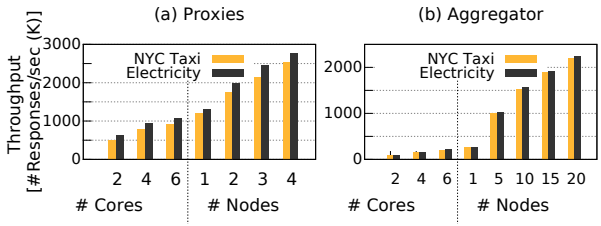


Figure 8: Throughput at proxies and the aggregator with different numbers of CPU cores and nodes.

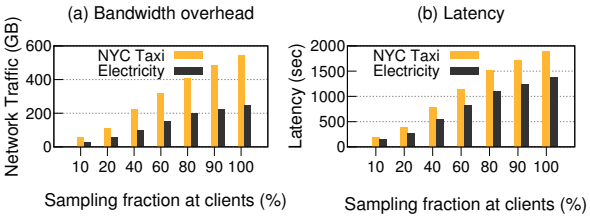


Figure 9: Total network traffic and latency at proxies with different sampling fractions at clients.

kWh, (0.5, 1] kWh, (1, 1.5] kWh, (1.5, 2] kWh, (2, 2.5] kWh, and (2.5, 3] kWh.

Evaluation metrics. We evaluated PRIVAPPROX using four key metrics: throughput, latency, utility, and privacy level. *Throughput* is defined as the number of data items processed per second, and *latency* is defined as the total amount of time required to process a certain dataset. *Utility* is the accuracy loss defined as $|\frac{estimate-exact}{exact}|$, where *estimate* and *exact* are the query results produced by applying PRIVAPPROX and the native computation, respectively. Finally, *privacy level* (ε_{zk}) is calculated using Equation 19 in the technical report [64]. For all measurements, we report the average over 10 runs.

7.2 Results from Case Studies

#I: Scalability. We measured the scalability of the two main system components: proxies and the aggregator. We first measured the throughput of proxies with different numbers of CPU cores (scale-up) and different numbers of nodes (scale-out). This experiment was conducted on a cluster of 4 nodes. Figure 8 (a) shows that, as expected, the throughput at proxies scales quite well with the number of CPU cores and nodes. In the NYC Taxi case study, with 2 cores, the throughput of each proxy is 512,348 answers/sec, and with 8 cores (1 node) the throughput is 1,192,903 answers/sec; whereas, with a cluster of 4 nodes each with 8 cores, the throughput of each proxy reaches 2,539,715 answers/sec. In the household electricity case study, the proxies achieve relatively higher throughput because the message size is smaller than in the NYC Taxi case study.

We next measured the throughput at the aggregator. Figure 8 (b) depicts that the aggregator also scales quite well when the number of nodes for aggregator increases. The throughput of the aggregator, however, is much lower than the throughput of proxies due to the relatively expensive `join` operation and the analytical computation at the aggregator. We notice that the throughput of the aggregator in the household electricity case study does not significantly improve in comparison to the first case study. This is because the difference in the size of messages between the two case studies does not affect much the performance of the `join` operation and the analytical computation.

#II: Network bandwidth and latency. Next, we conducted the experiment to measure the network bandwidth usage. By leveraging the sampling mechanism at clients, our system reduces network traffic significantly. Figure 9 (a) shows the total network traffic transferred from clients to proxies with different sampling fractions. In the first case study, with the sampling fraction of 60%, PRIVAPPROX can reduce the network traffic by 1.62×; whereas in the second case study, the reduction is 1.58×. Besides the benefit of saving network bandwidth, PRIVAPPROX also achieves lower latency in processing query answers by leveraging approximate computation. To evaluate this advantage, we measured the effect of sampling fractions on the latency of processing query answers. Figure 9 (b) depicts the latency with different sampling fractions at clients. For the first case study, with the sampling fraction of 60%, the latency is 1.68× lower than the execution without sampling; whereas, in the second case study, this value is 1.66× lower than the execution without sampling.

#III: Utility and privacy. Figure 7 (a)(b)(c) show the utility, the privacy level, and the trade-off between them,

respectively, with different sampling and randomization parameters. The randomization parameters p and q vary in the range of (0, 1), and the sampling parameter s is calculated using Equation 19 in the technical report [64]. Here, we show results only with the NYC Taxi dataset. As the sampling parameter s and the first randomization parameter p increase, the utility of query results improves (i.e., accuracy loss gets smaller) whereas the privacy guarantee gets weaker (i.e., privacy level gets higher). Since the NYC Taxi dataset is diverse, the accuracy loss and the privacy level change in a non-linear fashion with different sampling fractions and randomization parameters. Interestingly, the accuracy loss does not always decrease as the second randomization parameter q increases. The accuracy loss gets smaller when $q = 0.3$. This is due to the fact that the fraction of truthful "Yes" answers in the dataset is 33.57% (close to $q = 0.3$).

8 Related Work

Privacy-preserving analytics. Since the notion of differential privacy [32, 34], a plethora of systems have been proposed to provide differential privacy with centralized databases [46, 48, 52, 56–59, 63, 68]. In practice, however, such central trust can be abused, leaked, or subpoenaed [28, 49, 62, 69].

To overcome the limitations of the centralized database schemes, recently a flurry of systems have been proposed with a focus on preserving user privacy (mostly, differential privacy) in a distributed setting where the private data is kept locally [10, 26, 27, 33, 43, 44, 47, 55, 61, 71, 74]. However, these systems are designed to deal with the "one-shot" batch queries only, whereby the data is assumed to be static.

To overcome the limitations of the aforementioned systems, several differentially private stream analytics systems have been proposed [22, 23, 35, 38, 45, 66, 70]. Unfortunately, these systems still contain several technical shortcomings that limit their practicality. One of the first systems [35] updates the query result only if the user's private data changes significantly, and does not support stream analytics over an unlimited time period. Subsequent systems [23, 45] remove the limit on the time period, but introduce extra system overheads. Some systems [66, 70] leverage expensive secret sharing cryptographic operations to produce noisy aggregate query results. These systems, however, cannot work at large scale under churn; moreover, in these systems, even a single malicious user can substantially distort the aggregate results without detection. Recently, some other privacy-preserving distributed stream monitoring systems have been proposed [22, 38]. However, they all require some form of synchronization, and are tailored for heavy-hitter monitoring only. Streaming data publishing systems like [76] use a stream-privacy metric at the cost of relying on a trusted party to add noise. In contrast, PRIVAPPROX does not require a trusted proxy or aggregator to add noise. Furthermore, PRIVAPPROX provides stronger privacy properties (i.e., zero-knowledge privacy).

Sampling and randomized response. Sampling and randomized response, also known as input perturbation techniques, are being studied in the context of privacy-preserving analytics, albeit they are explored separately. For instance, the relationship between sampling and privacy is being investigated to provide k-anonymity [24], differential privacy [59], and crowd-blending privacy [40]. In contrast, we show that sampling combined with randomized response achieves the zero-knowledge privacy, a privacy bound strictly stronger than the state-of-the-art differential privacy.

Randomized response [37, 77] is a surveying technique in statistics, since 1960s, for collecting sensitive information via input perturbation. Recently, Google in a system called RAPPOR [73] made use of randomized response for privacy-preserving analytics. Like RAPPOR, PRIVAPPROX utilizes randomized response. However, RAPPOR is designed for heavy-hitter collection, and does not deal with the situation where clients' answers to the same query are changing over time. Therefore, RAPPOR does not fit well with the stream analytics. Furthermore, since we combine randomized response with sampling, PRIVAPPROX provides a privacy bound tighter than RAPPOR.

Approximate computation. Approximation techniques such as sampling [11, 25, 39], sketches [29], and online aggregation [50] have been well-studied over the decades in the databases community. Recently, sampling-based systems [8, 9, 42, 53, 65] have also been shown effective for "Big Data" analytics. We build on the advancements of sampling-based techniques. In particular, our work builds on IncApprox [53], a data analytics system that combines incremental computation [12, 15–18] and approximate computation. However, we differ in two crucial aspects. First, we perform sampling in a distributed way as opposed to sampling in a centralized dataset. Second, we extend sampling with randomized response for privacy-preserving analytics.

9 Conclusion

In this paper, we presented PRIVAPPROX, a privacy-preserving stream analytics system. Our approach builds on the observation that both computing paradigms — privacy-preserving data analytics and approximate computation — strive for approximation, and can be combined together to leverage the benefits of both. Our evaluation shows that PRIVAPPROX not only improves the performance to support real-time stream analytics, but also achieves provably stronger privacy guarantees than the state-of-the-art differential privacy. PRIVAPPROX's source code is publicly available: https://PrivApprox.github.io.

Acknowledgments. We thank anonymous reviewers and our shepherd Adam Bates for their helpful comments. This work is supported by the resilience path within CFAED at TU Dresden, the European Unions Horizon 2020 research and innovation programme under grant agreements 645011 (SERECA) and Amazon Web Services Education Grant.

References

[1] Apache S4. http://incubator.apache.org/s4. Accessed: May, 2017.

[2] Apache Spark Streaming. http://spark.apache.org/streaming. Accessed: May, 2017.

[3] Apache Storm. http://storm-project.net/. Accessed: May, 2017.

[4] Apache Zookeeper. https://zookeeper.apache.org/. Accessed: May, 2017.

[5] Kafka - A high-throughput distributed messaging system. http://kafka.apache.org. Accessed: May, 2017.

[6] Sample household electricity time of use data. https://goo.gl/0p2QGB. Accessed: May, 2017.

[7] SQLite. https://www.sqlite.org/. Accessed: May, 2017.

[8] Quickr: Lazily Approximating Complex Ad-Hoc Queries in Big Data Clusters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2016.

[9] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2013.

[10] I. E. Akkus, R. Chen, M. Hardt, P. Francis, and J. Gehrke. Non-tracking web analytics. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[11] M. Al-Kateb and B. S. Lee. Stratified Reservoir Sampling over Heterogeneous Data Streams. In *Proceedings of the 22nd International Conference on Scientific and Statistical Database Management (SSDBM)*, 2010.

[12] P. Bhatotia. *Incremental Parallel and Distributed Systems*. PhD thesis, Max Planck Institute for Software Systems (MPI-SWS), 2015.

[13] P. Bhatotia, U. A. Acar, F. P. Junqueira, and R. Rodrigues. Slider: Incremental Sliding Window Analytics. In *Proceedings of the 15th International Middleware Conference (Middleware)*, 2014.

[14] P. Bhatotia, M. Dischinger, R. Rodrigues, and U. A. Acar. Slider: Incremental Sliding-Window Computations for Large-Scale Data Analysis. Technical Report MPI-SWS-2012-004, MPI-SWS, 2012. http://www.mpi-sws.org/tr/2012-004.pdf.

[15] P. Bhatotia, P. Fonseca, U. A. Acar, B. Brandenburg, and R. Rodrigues. iThreads: A Threading Library for Parallel Incremental Computation. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[16] P. Bhatotia, R. Rodrigues, and A. Verma. Shredder: GPU-Accelerated Incremental Storage and Computation. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[17] P. Bhatotia, A. Wieder, I. E. Akkus, R. Rodrigues, and U. A. Acar. Large-scale incremental data processing with change propagation. In *Proceedings of the Conference on Hot Topics in Cloud Computing (HotCloud)*, 2011.

[18] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquini. Incoop: MapReduce for Incremental Computations. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2011.

[19] A. Blum, C. Dwork, F. McSherry, and K. Nissim. Practical privacy: the SuLQ framework. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, 2005.

[20] A. Blum, K. Ligett, and A. Roth. A Learning Theory Approach to Non-interactive Database Privacy. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 2008.

[21] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2015.

[22] T.-H. H. Chan, M. Li, E. Shi, and W. Xu. Differentially Private Continual Monitoring of Heavy Hitters from Distributed Streams. In *Proceedings of the 12th International Conference on Privacy Enhancing Technologies (PETS)*, 2012.

[23] T.-H. H. Chan, E. Shi, and D. Song. Private and Continual Release of Statistics. *ACM Trans. Inf. Syst. Secur.*, 2011.

[24] K. Chaudhuri and N. Mishra. When Random Sampling Preserves Privacy. In *Proceedings of the 26th Annual International Conference on Advances in Cryptology (CRYPTO)*, 2006.

[25] S. Chaudhuri, G. Das, and V. Narasayya. Optimized Stratified Sampling for Approximate Query Processing. *Proceedings of ACM Transaction of Database Systems (TODS)*, 2007.

[26] R. Chen, I. E. Akkus, and P. Francis. SplitX: High-performance Private Analytics. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, 2013.

[27] R. Chen, A. Reznichenko, P. Francis, and J. Gehrke. Towards Statistical Queries over Distributed Private User Data. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[28] ComScore Reaches $14 Million Settlement in Electronic Privacy Class Action. http://www.alstonprivacy.com/comscore-reaches-14-million-settlement-in-electronic-privacy-class-action/. Accessed: May, 2017.

[29] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Found. Trends databases*, 2012.

[30] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. Technical report, DTIC Document, 2004.

[31] J. R. Douceur. The Sybil Attack. In *Proceedings of 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[32] C. Dwork. Differential privacy. In *Proceedings of the 33rd International Colloquium on Automata, Languages and Programming, part II (ICALP)*, 2006.

[33] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor. Our Data, Ourselves: Privacy Via Distributed Noise Generation. In *Proceedings of the 24th Annual International Conference on The Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2006.

[34] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating Noise to Sensitivity in Private Data Analysis. In *Proceedings of the Third conference on Theory of Cryptography (TCC)*, 2006.

[35] C. Dwork, M. Naor, T. Pitassi, and G. N. Rothblum. Differential privacy under continual observation. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 2010.

[36] C. Dwork and A. Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[37] J. A. Fox and P. E. Tracy. *Randomized response: a method for sensitive surveys*. Beverly Hills California Sage Publications, 1986.

[38] A. Friedman, I. Sharfman, D. Keren, and A. Schuster. Privacy-Preserving Distributed Stream Monitoring. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2014.

[39] M. N. Garofalakis and P. B. Gibbon. Approximate Query Processing: Taming the TeraBytes. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2001.

[40] J. Gehrke, M. Hay, E. Lui, and R. Pass. Crowd-blending privacy. In *Proceedings of the 32nd Annual International Conference on Advances in Cryptology (CRYPTO)*, 2012.

[41] J. Gehrke, E. Lui, and R. Pass. Towards Privacy for Social Networks: A Zero-Knowledge Based Definition of Privacy. In *Theory of Cryptography*, 2011.

[42] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen. ApproxHadoop: Bringing Approximations to MapReduce Frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[43] S. Guha, B. Cheng, and P. Francis. Privad: Practical Privacy in Online Advertising. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[44] S. Guha, M. Jain, and V. N. Padmanabhan. Koi: A location-privacy platform for smartphone apps. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[45] T. h. Hubert Chan, E. Shi, and D. Song. Privacy-preserving stream aggregation with fault tolerance. In *Proceedings of 16th International Conference on Financial Cryptography and Data Security (FC)*, 2012.

[46] A. Haeberlen, B. C. Pierce, and A. Narayan. Differential Privacy Under Fire. In *Proceedings of the 20th USENIX Security Symposium (USENIX Security)*, 2011.

[47] M. Hardt and S. Nath. Privacy-aware personalization for mobile advertising. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*, 2012.

[48] M. Hay, V. Rastogi, G. Miklau, and D. Suciu. Boosting the Accuracy of Differentially Private Histograms Through Consistency. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2010.

[49] HealthCare.gov Sends Personal Data to Dozens of Tracking Websites. `https://www.eff.org/deeplinks/2015/01/healthcare.gov-sends-personal-data`. Accessed: May, 2017.

[50] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1997.

[51] Z. Jerzak and H. Ziekow. The debs 2015 grand challenge. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems (DEBS)*, 2015.

[52] V. Karwa, S. Raskhodnikova, A. Smith, and G. Yaroslavtsev. Private analysis of graph structure. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2011.

[53] D. R. Krishnan, D. L. Quoc, P. Bhatotia, C. Fetzer, and R. Rodrigues. IncApprox: A Data Analytics System for Incremental Approximate Computing. In *Proceedings of International Conference on World Wide Web (WWW)*, 2016.

[54] J. Lee and C. Clifton. How Much is Enough? Choosing ε for Differential Privacy. In *Proceedings of the 14th International Conference on Information Security (ISC)*, 2011.

[55] S. Lee, E. L. Wong, D. Goel, M. Dahlin, and V. Shmatikov. πBox: A Platform for Privacy-Preserving Apps. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[56] C. Li, M. Hay, V. Rastogi, G. Miklau, and A. McGregor. Optimizing Linear Counting Queries Under Differential Privacy. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, 2010.

[57] F. McSherry. Privacy Integrated Queries. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2009.

[58] F. McSherry and R. Mahajan. Differentially-private Network Trace Analysis. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, 2010.

[59] P. Mohan, A. Thakurta, E. Shi, D. Song, and D. Culler. GUPT: Privacy Preserving Data Analysis Made Easy. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2012.

[60] D. S. Moore. *The Basic Practice of Statistics*. W. H. Freeman & Co., 2nd edition, 1999.

[61] A. Narayan and A. Haeberlen. DJoin: Differentially Private Join Queries over Distributed Databases. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012.

[62] Privacy Lawsuit Targets Net Giants Over 'Zombie' Cookies. `http://www.wired.com/2010/07/zombie-cookies-lawsuit`. Accessed: May, 2017.

[63] D. Proserpio, S. Goldberg, and F. McSherry. Calibrating Data to Sensitivity in Private Data Analysis: A Platform for Differentially-private Analysis of Weighted Datasets. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2014.

[64] D. L. Quoc, M. Beck, P. Bhatotia, R. Chen, C. Fetzer, and T. Strufe. Privacy preserving stream analytics: The marriage of randomized response and approximate computing. `https://arxiv.org/abs/1701.05403`, 2017.

[65] D. L. Quoc, R. Chen, P. Bhatotia, C. Fetzer, V. Hilt, and T. Strufe. StreamApprox: Approximate Computing for Stream Analytics. 2017.

[66] V. Rastogi and S. Nath. Differentially private aggregation of distributed time-series with transformation and encryption. In

Proceedings of the International Conference on Management of Data (SIGMOD), 2010.

[67] M. G. Reed, P. F. Syverson, and D. M. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*, 1998.

[68] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and Privacy for MapReduce. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2010.

[69] SEC Charges Two Employees of a Credit Card Company with Insider Trading. `http://www.sec.gov/litigation/litreleases/2015/lr23179.htm`. Accessed: May, 2017.

[70] E. Shi, T. H. Chan, E. G. Rieffel, R. Chow, and D. Song. Privacy-Preserving Aggregation of Time-Series Data. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2011.

[71] K. Singh, S. Bhola, and W. Lee. xbook: Redesigning privacy control in social networking platforms. In *Proceedings of the 18th Conference on USENIX Security Symposium (USENIX Security)*, 2009.

[72] S. K. Thompson. *Sampling*. Wiley Series in Probability and Statistics, 2012.

[73] E. Úlfar, P. Vasyl, and K. Aleksandra. RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal Response. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[74] B. Viswanath, E. Kiciman, and S. Saroiu. Keeping Information Safe from Social Networking Apps. In *Proceedings of the ACM SIGCOMM Workshop on Social Networks (WOSN'12)*, 2012.

[75] G. Wang, B. Wang, T. Wang, A. Nika, H. Zheng, and B. Y. Zhao. Defending against sybil devices in crowdsourced mapping services. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2016.

[76] Q. Wang, Y. Zhang, X. Lu, Z. Wang, Z. Qin, and K. Ren. RescueDP: Real-time Spatio-temporal Crowd-sourced Data Publishing with Differential Privacy. In *Proceedings of the 35th Annual IEEE International Conference on Computer Communications (INFOCOM)*, 2016.

[77] S. L. Warner. Randomized response: A survey technique for eliminating evasive answer bias. In *Journal of the American Statistical Association*, 1965.

[78] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Brief Announcement: Modelling MapReduce for Optimal Execution in the Cloud. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of Distributed Computing (PODC)*, 2010.

[79] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Conductor: Orchestrating the Clouds. In *Proceedings of the 4th international workshop on Large Scale Distributed Systems and Middleware (LADIS)*, 2010.

[80] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the Deployment of Computations in the Cloud with Conductor. In *Proceedings of the 9th USENIX symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

Mercury: Bandwidth-Effective Prevention of Rollback Attacks Against Community Repositories

Trishank Karthik Kuppusamy Vladimir Diaz Justin Cappos
New York University Tandon School of Engineering

Abstract

A popular community repository such as Docker Hub, PyPI, or RubyGems distributes tens of thousands of software projects to millions of users. The large number of projects and users make these repositories attractive targets for exploitation. After a repository compromise, a malicious party can launch a number of attacks on unsuspecting users, including rollback attacks that revert projects to obsolete and vulnerable versions. Unfortunately, due to the rapid rate at which packages are updated, existing techniques that protect against rollback attacks would cause each user to download 2–3 times the size of an average package in metadata each month, making them impractical to deploy.

In this work, we develop a system called Mercury that uses a novel technique to compactly disseminate version information while still protecting against rollback attacks. Due to a different technique for dealing with key revocation, users are protected from rollback attacks, even if the software repository is compromised. This technique is bandwidth-efficient, especially when delta compression is used to transmit only the differences between previous and current lists of version information. An analysis we performed for the Python community shows that once Mercury is deployed on PyPI, each user will only download metadata each month that is about 3.5% the size of an average package. Our work has been incorporated into the latest versions of TUF, which is being integrated by Haskell, OCaml, RubyGems, Python, and CoreOS, and is being used in production by LEAP, Flynn, and Docker.

1 Introduction

Community repositories, such as Docker Hub [25], Python Package Index (PyPI) [69], RubyGems [71], and SourceForge [80], provide an easy way for third party developers to distribute software to users. Unlike traditional repositories (e.g., Ubuntu, or the Apple App Store), community repositories allow any developer to immediately release new software without waiting for an administrator's approval. This distinctive feature has led to the tremendous popularity of these repositories, which have served billions of downloads to millions of users.

Unfortunately, their popularity also makes them attractive targets for attackers. Major repositories run by Adobe, Apache, Debian, Fedora, FreeBSD, Gentoo, GitHub, GNU Savannah, Linux, Microsoft, npm, Opera, PHP, RedHat, RubyGems, SourceForge, and WordPress have all been compromised at least once [2–4, 21–24, 28, 31, 33, 34, 36, 44, 54, 59, 63, 70, 72, 81, 82, 86–88, 93].

When a community repository is compromised, a number of attacks can be launched on unsuspecting users, including *rollback attacks*, where attackers revert the state of the repository to point to obsolete and vulnerable versions of software. Rollback attacks are trivial for attackers to perform: instead of tampering with signed software, they simply replace these software packages with older versions. It is equally trivial to prevent such attacks for software that is already installed by the user, because existing security systems can easily reject software older than what is already on disk. However, there may be tens of thousands of software projects on a repository, of which the user may install only a fraction. Unless the user keeps track of all projects, she is susceptible to a rollback attack on a project she might install at a much later date. Consequently, she would install authentic but obsolete software that contains known vulnerabilities. An attacker can later exploit these vulnerabilities to compromise her machine.

A solution to prevent rollback attacks needs to meet several important properties in order to be adopted:

- *No administrative overhead.* There must not be additional servers to manage. Many community repositories are managed by volunteers that infrequently interact with the repository, and so the administrative burden must remain low.

- *Simple client communications.* Retrieving a package should not require clients to gossip or communicate with third parties. This could create deployment issues and even security concerns (e.g., informing untrusted parties which security fixes are being requested [13]).

- *Low overhead.* Repositories often have large bandwidth costs and use mirrors or CDNs to offload this burden. A solution must not substantially increase this cost, even if the repository hosts a large number of projects that are rapidly updated.

In this paper, we describe Mercury, a bandwidth-efficient system that prevents security attacks, including rollback attacks, even if a community repository is compromised. This work is innovative in providing low-bandwidth rollback protection. However, the main contribution of this work is how the insights behind Mercury can be used by real-world community repositories to solve a widespread problem. Mercury has been incorporated into the latest versions of TUF [47, 73], which is being integrated by Haskell [94], OCaml [32], RubyGems [75–77], Python [45, 46], and CoreOS [66], and is being used in production by LEAP [49], Flynn [68], and Docker [64].

The key insight in Mercury is that the source of trust about which versions of projects are current can be safely shifted from developers to the repository. The repository uses online keys to sign and distribute the latest version numbers of projects as soon as they are updated. Although attackers can provide clients with incorrect version information when a repository is compromised, Mercury uses several techniques that can limit user susceptibility to rollback attacks even in this case. The key technique is that by always comparing the current list of version information signed by the repository to the previous list, these attacks are easily detected. Mercury is bandwidth-efficient with respect to rollback attacks because it downloads only the version numbers of all projects (instead of metadata about all packages), and uses delta compression [61, 62] to transmit only the differences between previous and current lists. While trusting the repository for version numbers opens users up to a new fast-forward attack, this can be mitigated by performing additional steps when revoking the repository key after a compromise.

In summary, our **contributions** are:

1. We find that existing security systems that prevent rollback attacks incur prohibitive bandwidth costs to do so when the number of projects, or the rate of project updates, is high (e.g., in popular community repositories).

2. We design and implement Mercury, a bandwidth-efficient system that prevents rollback attacks even though it depends on the repository to continually indicate the latest versions of projects.

3. We evaluate the effectiveness of Mercury using requests to PyPI. We find Mercury can prevent rollback attacks by having each user download metadata each month that is about 3.5% the size of an average package. Additionally, new users (or all users following a compromise) will download metadata 48% of the size of an average package (compared to two other systems with overheads of 1,152% and 3,092%).

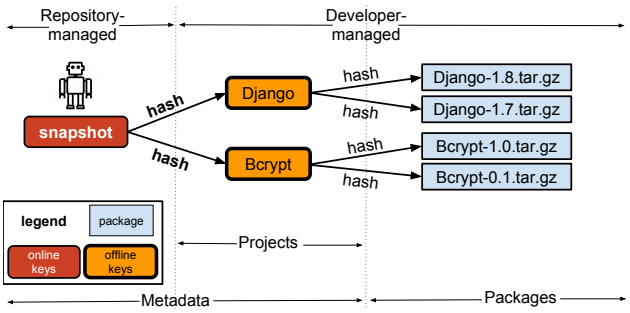


Figure 1: How software is organized by metadata about projects and packages on a community repository. Repositories and developers sign these metadata in order to prevent security attacks. See Section 2 for a detailed explanation.

2 Background

In order to better understand the design decisions behind Mercury, we provide some essential background information. First, we discuss how software is managed and distributed by community repositories. Then, we describe metadata used in an existing security system, TUF [47, 73], that we leverage in Mercury to protect these repositories from security attacks.

TUF is a framework that allows repositories to build different security models that provide varying degrees of security and usability. In this paper, we show that TUF has a severe performance drawback on popular community repositories. Thus, we devised Mercury, a more efficient variant of TUF that prevents rollback attacks using significantly lower bandwidth costs. As we stated earlier, Mercury has been incorporated into the latest versions of TUF.

2.1 Community repositories

A *community repository* is a single server that hosts and distributes third-party software. Three groups of people interact with the repository. *Administrators*, who are usually volunteers, manage the repository software and hardware. *Developers* upload software to the repository, which administrators publish as soon as possible, for users to download. *Users* download, validate, and

Django project metadata file


```
{
  "signatures": {
    "keyid": "4445d918dfcf1af804b749eeee4835dccfd27c06b6828533be827473ff63439f",
    "method": "ed25519",
    "sig": "190f4b6228f2f72b3cbafa3446e032c9eaed03b055acfe8a9d3c445060b47d1b3ccd5c1ed9a9367a53d21e1d265f453996268dfeb1f005e530a025b0676ec720"
  },
  "signed":{
    "packages": {
      "Django-1.7.tar.gz": "0654407104e420508cf5be04fb85a066131df3117117dbaca095e9a248949359"
      "Django-1.8.tar.gz": "066bad42cb4c66944e7efcf7304d3d17f7b0eb222e53958cdd866420d2e8b412"
    },
    "expires": "2015-03-21T00:00:00Z",
    "version": 2
  }
}
```


Figure 2: An example of a project metadata file for the repository in Figure 1, explained in Section 2.2.

Snapshot metadata file


```
{
  "signatures": {
    "keyid": "16a0eeb0791b6c92451fd284dd9f599e0a7dbe7f6ebea6e2d2d06c7f74aec112",
    "method": "ed25519",
    "sig": "7a7e4858a2f86f740c2a9d8627df4cda92f7b4b8e600ea596ffa3623ca31b0e7b0e59c3bd601645e03ae5ba0581d2c31a8ce3a879d34afdf09dc3040339bfac"
  },
  "signed":{
    "projects": {
      "Django.json": "1919ff5cc47994470e539169db049f61ff133538ea1b935484e9819e00beb9d6"
      "Bcrypt.json": "bd162a5385407e07e0e67310d8ebc60abe759d8937bf72ad1258024dff6f561a"
    },
    "expires": "2014-03-29T09:44:10Z"
  }
}
```


Figure 3: An example of a snapshot metadata file for the repository in Figure 1, explained in Section 2.2.

install software with a *package manager* that may download software through middlemen, such as content delivery networks (CDNs) and/or mirrors. These middlemen allow the repository to reduce bandwidth costs.

The software uploaded by developers is organized as follows. A developer registers a *project* with a unique name, such as `Django` or `Bcrypt`. When a specific version of the software for that project is ready to be released, the software is built into a *package* (e.g., `Django-1.7.tar.gz`), and the developer uploads that package to a community repository. A project may make multiple packages available at any time. For example, in Figure 1, even though `Django-1.8.tar.gz` may be the latest package of the `Django` project, `Django-1.7.tar.gz` is still available to users who request it.

2.2 Project and snapshot metadata

Appropriately structured and signed *metadata* can be used to prevent security attacks when a repository is compromised [15, 47, 73]. These metadata are used by package managers to tell whether attackers have tampered with projects, or reverted projects to obsolete versions. In this paper, we focus on two types of metadata.

Project metadata is the manifest of all packages released by a project [47]. It lists the cryptographic hashes for available packages, and includes an expiration date as well as a *version number* for the metadata file itself. In Figure 2, version 2 of the `Django` project metadata lists the hashes for the `Django-1.7.tar.gz` and `Django-1.8.tar.gz` packages, and an expiration date of March 21st 2015. Developers use *offline keys* (or private keys stored off the repository) to sign project metadata, so that attackers cannot modify it without being detected.

Snapshot metadata is the manifest of all project metadata currently available on the repository. Following common practice in traditional repositories [15], snapshot metadata binds the location (e.g., relative path) of every project metadata file to the cryptographic hash of the file [73]. In Figure 3, the snapshot metadata file lists the hashes for the `Django` and `Bcrypt` project metadata files. Since packages and project metadata are continually updated (as often as every few minutes [47]) and made available to users immediately, community repositories use *online keys* (or private keys stored on the repository) to sign snapshot metadata [47]. Because the snapshot key is stored on the repository, an attacker who compromises the repository can sign maliciously generated metadata with that key. In the next few sections, we discuss how Mercury deals with this scenario.

3 Threat model

In this paper, we are concerned with a scenario where attackers have compromised a community repository. Our threat model then assumes that:

1. Attackers can compromise a running repository, and tamper with any files and keys stored on the repository.

2. Developers store their keys off the repository, so that attackers cannot compromise these keys. Project metadata, which is managed and signed by the developers of each project, are not under the control of the attacker.

3. Attackers have access to any file that was previously published on the repository.

4. Attackers are aware of vulnerabilities in outdated packages, and are able to exploit them. These vulnerabilities can be found by looking at security an-

nouncements, or changes in source code repositories such as GitHub. However, attackers do not know of zero-day flaws in packages.

We leverage pre-existing techniques from TUF and other software security systems to provide effective protection against a wide array of other attacks [14, 16, 17, 47, 73]. As a result, our system can recover from key compromises [47, 73] and resist malicious man-in-the-middle attackers or mirrors [16]. Note that these techniques are orthogonal to Mercury, and do not interfere with its evaluation.

This work focuses on *rollback attacks* that cause package managers to install obsolete packages containing known vulnerabilities. A rollback attack happens when a package manager accepts a project that is older than the version at the last time the user visited the repository.

4 Analysis of the limitations of existing systems

A motivation for our work is that existing security systems that can be deployed on community repositories fall short for one of two reasons. They either do not prevent rollback attacks, or require prohibitive bandwidth costs to defend against such attacks.

4.1 Systems that are insecure

Many of the popular community repositories use either HTTPS or package signing (e.g., GPG or RSA) to ensure packages are not tampered with. This system does prevent rollback attacks on projects already installed by the user, because the package manager will not accept a project metadata file with a version number lower than in the previous copy of the file.

However, it suffers from a subtle but serious security problem. The package manager does not know about the version number of project metadata files for packages that are not requested by the user. If the repository is subsequently compromised, then attackers can execute rollback attacks on projects *yet to be installed* by the user. Hence, when an attacker compromises the repository, they can provide package managers out-of-date versions of packages that have known vulnerabilities.

4.2 Systems that are bandwidth-inefficient

As discussed in Section 2, Mercury is a variant of TUF [73], a security system deployed [47] by some community repositories. TUF protects users from rollback attacks by downloading developer-signed project metadata for all projects. This way, if a repository is compromised, the attacker cannot provide forged project metadata. To avoid detection, the package manager must be given project metadata that is at least as current as the previous project metadata downloaded by the package manager.

This system prevents rollback attacks on projects yet to be installed by the user, but has high bandwidth costs in two cases. First, for any new user (i.e., a user that has no previous project metadata), the package manager must download all project metadata files on the repository. This may be large since there may be tens of thousands of projects and hundreds of thousands of packages. Second, projects are continually created or updated on community repositories. Thus, returning users will download significant amounts of metadata to update to the latest version. As a result, this security system can be costly. However, the bandwidth cost for TUF is low should users need to recover from a repository compromise. Since the developer signs all of the project metadata, it need not be revoked if an attacker controls the repository. So, recovery from compromise is inexpensive, while normal operation is costly.

While it is not used in practice, for comparison purposes we also propose TUF-version, a variant of TUF where a project developer separately signs a *project-version metadata* file that simply contains the version number of her project. Then, the package manager downloads all project-version metadata files, but only the project metadata file for the package to be installed. The number of signatures is a significant cost for the project-version metadata. Thus, as we will see later in Section 6, this variant incurs between 37–53% of the cost of TUF, but is still too expensive for community repositories.

5 Mercury: a new security system

To address the limitations of existing security systems, we present Mercury, a security system for community repositories that can prevent rollback attacks while using a reasonable amount of bandwidth. Mercury retains security even if a potentially compromised repository signs version information on behalf of all projects. This is due to its slightly more complicated functionality when recovering from a repository compromise. Thus the "rare case" of recovering from a compromise is less straightforward, but the "common case" of distributing version information requires much less bandwidth. In this section, we discuss how and why package managers using Mercury will be protected from rollback attacks.

5.1 Insight: shifting trust from developers to the repository

Existing systems (Section 4.2) are expensive because they were designed with the assumption that there is no

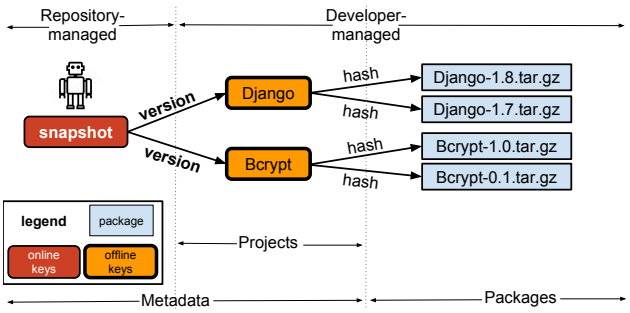


Figure 4: In Mercury, the snapshot metadata binds the location of every project metadata file to the *version number* instead of the hash of the file.

Snapshot metadata file

```
{
 "signatures": {
  "keyid": "16a0eeb0791b6c92451fd284dd9f599e0a7dbe7f6ebea6e2d2d06c7f74aec1
           12",
  "method": "ed25519",
  "sig": "7a7e4858a2f86f740c2a9d8627df4cda92f7b4b8e600ea596ffa3623ca31b0e7b
          0e59c39bd601645e03ae5ba0581d2c31a8ce3a879d34afdf09dc3040339bfac"
 },
 "signed":{
  "projects": {
   "Django.json": "2"
   "Bcrypt.json": "1"
  },
  "expires": "2014-03-29T09:44:10Z"
 }
}
```

Figure 5: An example of a Mercury snapshot metadata file for the repository illustrated in Figure 4. See Section 5.1 for details.

trusted party (e.g., hardware or administrators) on the repository that can always correctly indicate the version numbers of project metadata files. In the absence of this trusted party, package managers have relied on project metadata files signed by developers to learn about version numbers, even though it has meant downloading all new files.

Our key insight is that by handling key revocation in a different manner (Section 5.3), a repository can securely distribute the version numbers of project metadata files in the snapshot metadata. In Mercury, the snapshot metadata binds the location of every project metadata file to its *version number* instead of the hash of the file (as illustrated in Figure 4 and Figure 5). Now, the snapshot metadata informs the package manager not only about which projects on the repository are new or updated, but also gives the version numbers of their corresponding project metadata files. By shifting the source of trust from developers to the repository, Mercury allows the package manager to save bandwidth as long as it: (1) has access to a previous snapshot metadata file that was signed by the repository, and (2) always verifies the current snapshot metadata file as follows.

Suppose the user wishes to install a `Django` package. The package manager begins by downloading the difference between the previous and current snapshot metadata files, s_{prev} and s_{curr}, respectively. Next, the package manager must verify that the version number b of every project metadata file in s_{curr} is greater than or equal to the version number a of the same project metadata file in s_{prev}. If this verification step passes, then it sets s_{prev} to s_{curr}. Finally, the package manager downloads only the `Django` project metadata file, and ensures that the version number c in this file is indeed equal to the version number b for this file in s_{curr}.

There are two reasons why this saves bandwidth cost. First, the package manager downloads only a new project metadata file for the package to be installed, as opposed to all new project metadata files. Second, in Mercury there is a single signature (from the repository) in a snapshot metadata file. With TUF / TUF-version, the package manager downloads all new or updated project / project-version metadata files (and hence metadata about their packages).

5.2 Security analysis

A primary strength of Mercury is that an attacker who compromises the repository cannot rollback projects to versions that existed before the last time the user visited it. This is because whenever the user installs a package, the package manager always compares the current snapshot metadata file s_{curr} to the previous copy s_{prev}. The package manager would detect a rollback attack, and refuse to install the package, if: (1) the version number b of any project metadata file in s_{curr} is lower than the version number a of the same project metadata file in s_{prev}, or (2) the version number c of the project metadata file for the requested package is lower than the version number b for this project metadata file in s_{curr}.

As with existing systems [47,73], the attacker can rollback projects to versions that were added after the last time the user visited the repository. However, unlike existing systems, Mercury provides a stronger method for imposing stringent limits on these attacks (Section 5.4).

Attackers can deny the installation of packages by executing *fast-forward attacks*, where they arbitrarily increase the version numbers of project metadata files in the snapshot metadata. In a sense, fast-forward attacks are the opposite of rollback attacks. In this attack, the version number b of at least one project metadata file in s_{curr} is greater than the version number a of the same project metadata file in s_{prev}. However, this version number b is also greater than the actual version number c contained within the project metadata file itself. Thus,

the package manager would refuse to install a package from this project.

Fast-forward attacks are not nearly as severe a threat as rollback attacks because they simply block a package from being installed. Since the attacker has multiple ways to achieve the same goal (the simplest of which is to refuse to serve anything), fast-forward attacks do not present a major threat so long as it is possible to recover from them securely.

5.3 Recovering from a repository compromise

As discussed earlier, attackers who compromise a repository may launch fast-forward attacks that prevent the user from installing newer versions of existing software. This problem can be addressed by replacing the package manager's copy of the snapshot metadata. To do so, administrators must use an offline backup [47] to restore all project metadata and packages to a verifiable point before the compromise. Then, the online keys used to sign snapshot metadata can be revoked and replaced with new keys.

The process for distributing and revoking these keys is borrowed from TUF [47, 73]. The repository signs root-of-trust metadata using a quorum of offline keys. The root-of-trust metadata indicates which keys can be trusted for verifying metadata files, including snapshot metadata files. This leads to a seamless and automatic recovery from fast-forward attacks after a repository compromise.

5.4 Securing out-of-date package managers

The security of a Mercury user relies on her package manager possessing version numbers that are relatively recent. Users who have never visited the repository before are protected against rollback attacks by bundling the latest root-of-trust and snapshot metadata with the package manager. Nevertheless, a package manager using Mercury is vulnerable to rollback attacks against software released after the last time the package manager was updated. (Note that this limitation also applies to TUF for the same reason.) To combat this, a repository can choose to periodically sign a version of the snapshot metadata using offline keys (see s_{mid} in Figure 6). For example, if the repository administrator commits to signing snapshot metadata with offline keys at least every month, then the package manager can first retrieve that snapshot metadata, and verify that it was signed within the last month. Then, it verifies that all version numbers in the snapshot metadata signed with the online keys are later than or equal to those signed with the offline keys. This

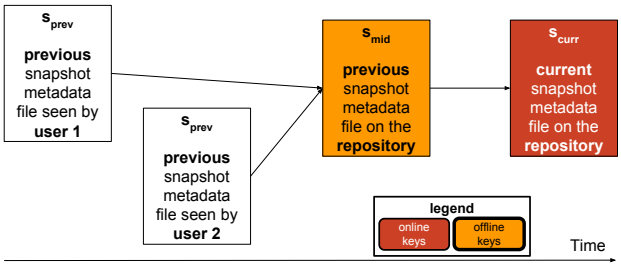


Figure 6: In order to help outdated package managers catch up to the latest snapshot metadata signed before the repository is compromised, administrators may periodically sign a copy s_{mid} of the latest snapshot metadata using offline keys. See Section 5.4 for details.

prevents attackers who compromised the repository from blocking packages that were released in the last month.

However, this functionality is not used in production by current users of Mercury. This is largely due to two concerns. First, the management overhead of having separate keys stored securely offline was deemed high for this use case. Second, there was some concern that the administrator would forget to sign an update with the offline keys within the prescribed period, and that this would cause users to incorrectly deduce an attack was underway. Hence, Docker [64] and Flynn [68] do not use this feature of Mercury in their deployments.

5.5 Deleting projects from snapshot metadata

It is fairly common practice for community repositories to allow projects to be deleted. However, deleting projects can make it harder for Mercury to defend against rollback attacks. Suppose that the package manager naively drops the version information for projects deleted from the snapshot metadata file. This would enable an attacker who controls the repository to reset known version numbers. Therefore, to better secure a repository using Mercury, projects should not be deleted from snapshot metadata. This is the route that Docker [64] and Flynn [68] chose with their deployments.

5.6 Protection against malicious mirrors

Some community repositories, such as Docker [64], use mirrors to serve metadata and packages to users, which opens users to malicious mirrors that may be able to tamper with some files. Specifically, consider a scenario outside of our threat model, where malicious mirrors do not have access to snapshot metadata keys, but have access to a few keys used to sign some project metadata files. These mirrors cannot tamper with the snapshot metadata.

However, they can substitute a few original project metadata files with malicious project metadata files. These malicious project metadata files contain version numbers identical to original project metadata files, but point to malicious instead of original packages. Mercury cannot detect these substitutions, because there is only information about the version numbers of project metadata files in its snapshot metadata.

In order to address this problem, we propose Mercury-hash, a variant of Mercury where the snapshot metadata contains both version numbers and hashes of all project metadata files. This prevents a malicious mirror from substituting project metadata files without being detected. As we will see in Section 6, this variant incurs 7x the cost of Mercury, which may be acceptable for community repositories where preventing this problem is important.

5.7 Implementation

Our reference implementation of Mercury is written in Python. It includes: command-line tools [89, 90] that help administrators and developers create, sign, and validate metadata (4,661 SLOC); integration libraries that package managers can use to download and verify metadata as well as packages (1,218 SLOC); unit and integration tests (6,247 SLOC); documentation such as specifications, and example metadata.

6 Evaluation of bandwidth costs

In the previous section, we discussed how Mercury is designed to prevent rollback attacks, even if the repository is compromised. In this section, we show that very same design is also efficient with respect to bandwidth cost. Using a log of package downloads from PyPI, the Python community repository, we compare Mercury to existing security systems, and answer the following questions:

1. What is the bandwidth overhead needed by each security system to prevent rollback attacks on PyPI? (Section 6.2)

2. How does the bandwidth overhead change as the number of projects on PyPI is varied? (Section 6.3)

3. How does the bandwidth overhead change as the rate of project updates on PyPI is varied? (Section 6.4)

6.1 Experimental setup

To answer these questions, we obtained an anonymized log of package downloads from PyPI for the month between March 21st and April 19th, 2014. This log contains 69,890,162 package requests by 1,175,625 users (identified by anonymized IP addresses). These users downloaded 46TB of packages, and the average downloaded package size was 660KB. We elected to use the average downloaded package size as a basis of comparison for metadata overhead, because it is the average expenditure when obtaining new or updated software packages. As such, it serves as a logical frame of reference in determining whether metadata overhead is reasonable or excessive.

To measure the cost for a package request in the download log, we must know the file sizes of packages and their corresponding metadata. To obtain package file sizes, we copied all packages hosted on PyPI at the time of writing. (Thus, these are approximations of the file sizes of packages available that month.) To obtain metadata file sizes, we produced 10,981 *releases* of metadata as follows. The first release contains snapshot, project-version, and project metadata about all packages that were available at the beginning of the month. Then, we produced a new release whenever a project was created or updated during the month. The first and last releases describe 58,328 and 59,486 projects, respectively. When computing the cost for a request, we compared using compression, delta encoding [61, 62], or delta compression [41, 42], and chose the most cost-efficient method.

We compare Mercury to four security systems that have been, or can be, deployed by community repositories (Section 4). One security system does not prevent rollback attacks (Section 4.1). In many deployments of this system, used by community repositories such as PyPI and RubyGems, developers use GPG or RSA to sign project metadata. Thus, as a useful abbreviation, we will call this security system GPG/RSA. We also compare Mercury against a variant called Mercury-hash (Section 5.6). Finally, we compare Mercury against TUF [47, 73] and a variant called TUF-version (Section 4.2).

The source code and data for these experiments are freely available at `https://theupdateframework.com/`. Unfortunately, the download log is not publicly available, because it may inadvertently compromise the privacy of PyPI users.

6.2 Bandwidth overhead by security system

The initial benchmark required in our study was the bandwidth cost for all five systems. This was measured by looking at the cost per user. A user may incur three different types of costs. First, a new user who just installed the PyPI package manager incurs an *initial cost* to download its first copy of metadata. Second, a user

	Initial cost	Recurring cost	Recovery cost
GPG/RSA	0.6KB (0.1%)	0.02KB (0.003%)	N/A
Mercury	319KB (48%)	23KB (3.5%)	320KB (48%)
Mercury-hash	2.4MB (360%)	156KB (24%)	2.4MB (361%)
TUF-version	7.6MB (1,152%)	1.1MB (171%)	2.3MB (350%)
TUF	20MB (3,092%)	2.1MB (320%)	2.3MB (350%)

Table 1: The overhead for a user incurred by each security system. The user incurs an initial cost when she contacts PyPI for the first time, a recurring cost when she returns to an uncompromised PyPI after the month of the download log, and a recovery cost when she returns to PyPI after it has recovered from a compromise. The percentages indicate the overhead relative to the average downloaded package size.

	Total initial costs of new users
Packages	2.2TB
GPG/RSA	0.005TB (0.2%)
Mercury	0.4TB (17%)
Mercury-hash	2.8TB (125%)
TUF-version	8.9TB (396%)
TUF	23.9TB (1,067%)

Table 2: The overhead to PyPI incurred by each security system for new users. We consider every IP address that appears for the first time in the download log as a new user. The percentages indicate the overhead relative to the total size of all packages downloaded by these new users.

who returns to an uncompromised PyPI incurs a *recurring cost* to update the metadata (which is the common case). Third, a user who returns to PyPI after it has recovered from a compromise incurs a *recovery cost* to re-download metadata to deal with the compromise.

Table 1 lists these costs. The first column shows the initial cost. The second column shows the recurring cost after the month represented by our download log. We chose this period in order to study the greatest recurring cost that can be measured with the available data. The third column shows the recovery cost.

If she is a new user, then GPG/RSA incurs the lowest initial cost (0.1%) relative to the average downloaded package size. (The cost is equal to the average size of project metadata files available in the last release.) This is because it downloads only the project metadata file for the requested package. However, the user is left vulnerable to rollback attacks against all other projects. Mercury incurs a larger initial cost (48%), because it must also download snapshot metadata about all projects. However, this is a one-time cost, and protects the user from rollback attacks against all known projects. Mercury-hash incurs an even larger initial cost (360%), because its snapshot metadata also contains the hashes of all project metadata files. In contrast, TUF-version (1,152%) and TUF (3,092%) incur significantly higher initial costs for the same protection. This is because they must download all project-version and project metadata, respectively. TUF-version is still bandwidth-inefficient compared to Mercury and even Mercury-hash, because it downloads incompressible signatures for nearly sixty thousand projects.

If she is returning to an uncompromised PyPI after the month, then GPG/RSA again incurs the lowest recurring cost (0.003%), because it downloads only the difference to the project metadata file for the requested package. (This recurring cost is equal to the average size of differences to project metadata files between the first and last releases.) Now, Mercury incurs significantly less recurring cost (3.5%), because it needs to download only the difference to snapshot metadata over the month. Mercury-hash incurs a larger recurring cost (24%), because it needs to also download hashes in its snapshot metadata. In contrast, TUF-version (171%) and TUF (320%) still incur a recurring cost greater than the average downloaded package, because they must download project-version and project metadata, respectively, about all projects created or updated over the month.

If she is returning to PyPI after it has recovered from a compromise, then she may incur a recovery cost for re-downloading metadata. GPG/RSA does not have a recovery cost (N/A), because there is no recovery procedure to reset the version numbers of projects after a repository compromise. On the other hand, Mercury has a recovery cost (48%) that is dominated by snapshot metadata (for the reason explained in Section 5.3). Similarly, the recovery cost (350%) for TUF-version and TUF are dominated by snapshot metadata. The snapshot metadata in both systems is always a few times larger than in Mercury, because it lists random hashes, instead of version numbers. However, note that, in this instance, the recovery cost for Mercury-hash is the largest, because its snapshot metadata contains both hashes and version numbers.

Finally, for the sake of completeness, we also look at the cost to PyPI incurred by "new" users in this month, or users who appear for the first time in our download log. We do not have ground truth about the number of new versus returning users, since that information cannot be determined from our download log. However, we can get a conservative estimate of this size, by assuming all users are new. While this may overestimate the cost to PyPI in this month, this is accurate for many virtualized environments, such as continuous integration / deployment systems. Table 2 lists these costs. New users downloaded 2.2TB of packages, and GPG/RSA adds to this an overhead of 0.005TB (0.2%) in project metadata. Mercury and Mercury-hash add an overhead of 0.4TB (17%) and 2.8TB (125%), mostly due to snapshot meta-

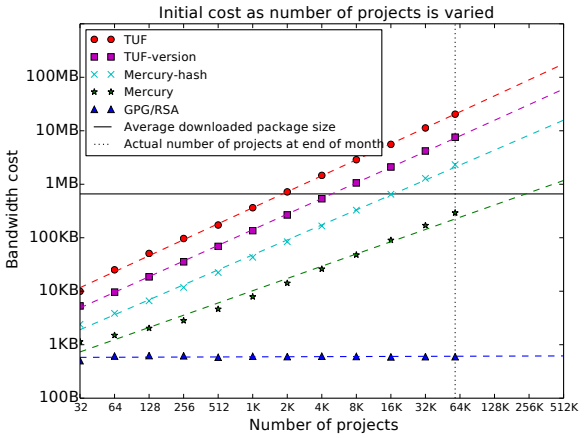


Figure 7: The initial cost for a user incurred by each security system, depending on the number of projects. The dashed lines show the regression lines based on the observed data (points) for each system. The horizontal line marks the average downloaded package size, whereas the vertical line marks the actual number of projects on PyPI at the end of the month. Points on the x and y axes have been plotted on the log-2 and log-10 scales, respectively.

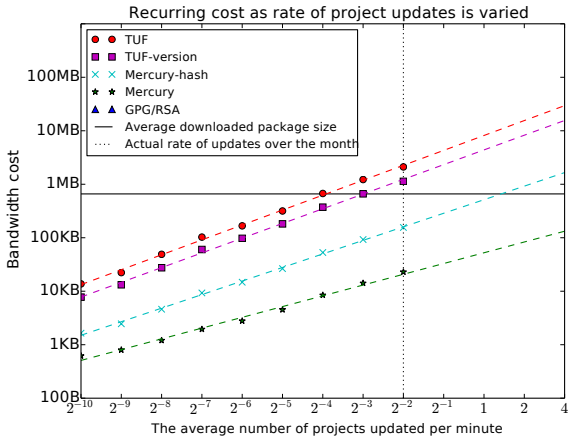


Figure 8: The recurring cost for a user incurred by each security system, if she returns to an uncompromised PyPI after the month of the download log, depending on the rate of project updates. The dashed lines show the regression lines based on the observed data (points) for each system. The horizontal line marks the average downloaded package size, whereas the vertical line marks the actual rate of project updates on PyPI over the month. Points on the x and y axes have been plotted on the log-2 and log-10 scales, respectively. Note that GPG/RSA is not represented in this figure, because its average recurring cost is effectively zero, since most projects were not updated during that month.

data. TUF-version adds an overhead of 8.9TB (396%), more than two-thirds of which is due to project version metadata. TUF adds an overhead of 23.9TB (1,067%), eighty-nine percent of which is due to project metadata. It is fair to conclude that it is not practical for community repositories to deploy TUF, or even TUF-version, especially when Mercury, and even Mercury-hash, can prevent rollback attacks just as well, but at a fraction of the cost.

6.3 Bandwidth versus number of projects

This subsection focuses on how the bandwidth costs would vary for a repository that has fewer or more projects than PyPI did at the end of the month. To answer this, we looked at how the initial cost for a new user would change as the number of projects in the last release (which contains the largest number of projects) is varied. (We focused on this cost because changing the number of projects on the repository would affect new users the most.) To study the cost if the number of projects is smaller than in the last release, we produced a new release based on a random sample of projects. On the other hand, to study the cost if the number of projects is larger than in the last release, we used linear regression to extrapolate the costs for this number based on the costs for smaller numbers of projects.

Figure 7 shows these costs. The vertical line marks the number of projects at the end of the month. The bandwidth overhead at that number of projects for all security systems is similar to the first column of Table 1. The horizontal line marks the average downloaded package size.

The initial cost for GPG/RSA changes little as the number of projects is varied, because its cost depends only on the size of the average project metadata file. In contrast, the initial costs for Mercury, Mercury-hash, TUF-version, and TUF grow linearly with the number of projects. With Mercury, this cost is dominated by the snapshot metadata. It outgrows the average downloaded package if the number of projects on PyPI grows larger by more than 4x (256K). The cost for Mercury-hash is also dominated by the snapshot metadata. However, it outgrows the average downloaded package if the number of projects on PyPI is nearly 3.4x smaller (17K).

Unlike Mercury, the costs for TUF-version and TUF are dominated by project-version and project metadata files, respectively. In fact, the cost for TUF-version is already greater than the average downloaded package if the number of projects on PyPI is nearly 12x smaller (5K), and for TUF if this number is more than 29x smaller (2K).

6.4 Bandwidth versus rate of project updates

This last subsection focuses on how the bandwidth costs would vary for a repository that has a lower or higher rate of project updates than PyPI did over the month. To answer this, we looked at how the recurring cost for a returning user would change as the rate of project updates varies between the beginning and end of the month. (We focused on this cost because changing this rate would principally affect users who are returning to an uncompromised repository.) Between these two points, 3,612 projects were created or updated 10,980 times. To study the cost if this rate is decreased, we artificially decreased it by increasing the time interval between the first and any subsequent release. For example, say that there are only three releases, and that the second and last releases were produced n and $2n$ minutes, respectively, after the first release. Since we assume that this user returns to the repository at the end of the month (say, at $2n+1$ minutes), her security system would download metadata from the last release. To artificially slow down the rate of project updates by half, the second and last releases would arrive $2n$ and $4n$ minutes, respectively, after the first release. Now, her security system would download metadata from the second release instead of the last one. On the other hand, to study the cost if this rate is increased, we used linear regression to extrapolate the costs for larger rates based on the costs for smaller rates.

Figure 8 shows these costs. The vertical line marks the actual rate of project updates at the end of the month (2^{-2} projects per minute, or a project every 4 minutes). The bandwidth overhead then for all security systems is identical to the second column of Table 1. The horizontal line marks the average downloaded package size. Note that GPG/RSA is not represented in this figure, because its average recurring cost is effectively zero, since most projects were not updated during that month.

When the rate of project updates is varied, the cost for Mercury and Mercury-hash are determined by the differences to snapshot metadata as projects are created or updated. However, the cost for Mercury remains well under the average downloaded package even if the rate of project updates is 16x higher than on PyPI (4 projects per minute, or 16 projects every 4 minutes). The cost for Mercury-hash is only greater than the average downloaded package when the rate of project updates is nearly 5.7x higher than on PyPI (1.4 projects per minute, or 5.6 projects every 4 minutes). In contrast, the cost for TUF-version is already greater than the average downloaded package when the rate of project updates is 2x lower than on PyPI (2^{-3} projects per minute, or a project every 8 minutes), and for TUF when this rate is 4x lower (2^{-4} projects per minute, or a project every 16 minutes).

7 Related work

In this section, we survey some prior work that is related to Mercury.

Accountability systems. Accountability systems, such as PeerReview [37], CATS [95], and CloudProof [67], provide a way to detect a subclass of Byzantine failures in distributed systems. All of these systems can detect rollback attacks after they happen, but, unlike Mercury, they are not designed to prevent such attacks before they occur.

Security systems for software repositories. Previous work have shown software updaters to be prone to security problems such as rollback attacks [5, 15]. Popular Linux package managers use a security architecture that protects against malicious mirrors or CDNs [15]. But, unlike Mercury, it will not necessarily withstand a compromise of the original repository [47, 73].

Revere [53] uses a self-organizing, peer-to-peer (P2P) overlay network to deliver security updates. However, a P2P setup would increase the complexity of deploying a community repository, and as such, was deemed impractical by the administrators we have been working with. Since Mercury does not require a P2P setup, it is an easier system to put in place.

File systems for untrusted storage servers. In this subsection, we will discuss a number of file systems that are inherently designed to detect whether attackers have tampered with packages. The biggest difference is that Mercury is not a file system, which means that repositories are free to use any file system that they like. Mercury works on top of existing file systems, and requires repositories only to add a layer of signed metadata, and modifying package managers to verify these metadata before installing packages.

ECFS [6] and TCFS [19], both of which are based on the Cryptographic File System (CFS) [7], allow developers to share files with users by offering the option of not encrypting files. However, ECFS does not appear to prevent rollback attacks on files not yet read by the user, whereas TCFS does not prevent rollback attacks at all, because unencrypted files are not protected with digital signatures. By providing security without the need to encrypt, Mercury offers a more accessible alternative.

To guarantee freshness, SiRiUS [35] requires every project developer to sign a hash tree of metadata files. This signature expires quickly, and so a software agent acting on behalf of the developer must renew it every few seconds or minutes. Unfortunately, this would not work on community repositories that provide rarely updated projects which are still heavily used, but no longer actively maintained by developers [47]. Unlike SiRiUS, Mercury does not require developers to quickly renew signatures on project metadata.

SNAD [60] can prevent rollback attacks against all projects by using a certificate object, which serves a similar purpose to the snapshot metadata in Mercury. However, SNAD is computationally expensive for community repositories, because all files must be encrypted, even though these repositories have no need for encryption.

The Protected File System (PFS) [84] records hashes of file blocks, where each hash is parameterized not only with the file block itself, but also a secret key kept on trusted storage. This prevents attackers from tampering with blocks. However, like Iris [83], PFS assumes that both developers and users would share the same secret key to read and write files. Sharing this secret key only makes sense when the users share the same computer, as in PFS, or the same organization, as in Iris. Mercury does not require developers or users to share a secret key, which means that they do not have to share the same computer or organization.

Security systems with different trust assumptions. SUNDR [51, 57] is a file system designed for software repositories. Unlike Mercury, SUNDR can prevent rollback attacks as well as detect forking attacks [8–12, 78] despite using a single untrusted server. However, the price of this is that SUNDR requires clients to trust that other clients would honestly report whether the repository has performed a forking attack. The problem is that a single faulty or malicious client could accidentally or deliberately frame an honest repository.

Depot [55] is a file system that eliminates trust for safety, and minimizes trust for liveness and availability. Unlike Mercury, Depot not only detects forking attacks, but can continue functioning despite these attacks. However, the price is potentially high bandwidth costs, because Depot is essentially a replication protocol that requires clients to continually exchange updates about all read and write operations with servers or other clients.

In the most popular method used in file systems to provide file integrity, a trusted party signs a Merkle hash tree [58] over a set of files [27,29,30,35,39,40,56,65,83, 92]. Unfortunately, there is no such trusted party on community repositories. Community repositories must use online keys instead to sign the root of this tree, because packages are continually updated, and must be published as soon as possible (Section 2.2). Unfortunately, attackers who compromise the repository can use these online keys to sign new hash trees that point to obsolete project metadata files. Mercury does not use hash trees, and addresses this problem by distributing version numbers of all project metadata files using the snapshot metadata, which help to prevent rollback attacks.

Other systems, such as Proof of Freshness [92], A2M [20], and TrInc [50], assume that there is trusted hardware (such as a Trusted Platform Module [91] chip). Unfortunately, except in limited settings [85], such trusted hardware is generally not available on commodity cloud servers that community repositories may use to host packages [1]. Mercury does not need trusted hardware, which greatly increases where it can be deployed.

Byzantine fault-tolerant security systems. Byzantine fault-tolerant (BFT) systems use many replicas instead of a single server to execute operations [18, 26, 38, 43, 52, 74, 79]. Unfortunately, PBFT requires administrators to manage $3f + 1$ independent replicas instead of a single server [48], where f is the maximum number of repositories whose compromise can be tolerated. This significantly increases administrative burden. Mercury can work using only a single server, making it less expensive, and more easily deployable.

8 Conclusions

As community repositories continue to grow in popularity, so does the need for reliable and economically-feasible security systems to protect users from a number of possible attacks. Solutions that require developers to indicate the latest version number are too costly to be used in practice. In this paper, we present Mercury, a security system that instead uses the community repository to indicate the latest version numbers of projects. Although attackers can compromise the repository, Mercury always prevents rollback attacks, and its recovery mechanism helps users recover from fast-forward attacks. Using a key on the repository to sign the version number for every project allows Mercury to efficiently use bandwidth to prevent rollback attacks.

The Mercury source code and standards documents are freely available at `https://theupdateframework.com/`.

Acknowledgements

We thank our shepherd, Eric Eide, and the anonymous reviewers for their valuable comments. We would also like to thank Lois Anne DeLong for her efforts on this paper, as well as the Docker, CoreOS, Flynn, Haskell, LEAP, OCaml, Python, Ruby, and Square communities for their collaboration. Our work on Mercury was supported by U.S. National Science Foundation grants CNS-1345049 and CNS-0959138.

References

[1] ACHEMLAL, M., GHAROUT, S., AND GABER, C. Trusted Platform Module as an Enabler for Security in Cloud Computing. In *Network and Information Systems Security (SAR-SSI), 2011 Conference on* (May 2011), pp. 1–6.

[2] APACHE INFRASTRUCTURE TEAM. apache.org incident report for 8/28/2009. https://blogs.apache.org/infra/entry/apache_org_downtime_report, 2009.

[3] APACHE INFRASTRUCTURE TEAM. apache.org incident report for 04/09/2010. https://blogs.apache.org/infra/entry/apache_org_04_09_2010, 2010.

[4] ARKIN, B. Adobe to Revoke Code Signing Certificate. https://blogs.adobe.com/conversations/2012/09/adobe-to-revoke-code-signing-certificate.html, 2012.

[5] BELLISSIMO, A., BURGESS, J., AND FU, K. Secure Software Updates: Disappointments and New Challenges. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Security* (Berkeley, CA, USA, 2006), HOTSEC'06, USENIX Association, pp. 7–7.

[6] BINDEL, D., CHEW, M., AND WELLS, C. Extended Cryptographic File System, 1999. Unpublished. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.4339.

[7] BLAZE, M. A Cryptographic File System for UNIX. In *Proceedings of the 1st ACM Conference on Computer and Communications Security* (New York, NY, USA, 1993), CCS '93, ACM, pp. 9–16.

[8] CACHIN, C., AND GEISLER, M. *Applied Cryptography and Network Security: 7th International Conference, ACNS 2009, Paris-Rocquencourt, France, June 2-5, 2009. Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, ch. Integrity Protection for Revision Control, pp. 382–399.

[9] CACHIN, C., KEIDAR, I., AND SHRAER, A. Fork sequential consistency is blocking. *Information Processing Letters 109*, 7 (2009), 360–364.

[10] CACHIN, C., KEIDAR, I., AND SHRAER, A. Fail-Aware Untrusted Storage. *SIAM Journal on Computing 40*, 2 (2011), 493–533.

[11] CACHIN, C., AND OHRIMENKO, O. *Principles of Distributed Systems: 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings*. Springer International Publishing, Cham, 2014, ch. Verifying the Consistency of Remote Untrusted Services with Commutative Operations, pp. 1–16.

[12] CACHIN, C., SHELAT, A., AND SHRAER, A. Efficient Fork-linearizable Access to Untrusted Shared Memory. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2007), PODC '07, ACM, pp. 129–138.

[13] CAPPOS, J. Avoiding Theoretical Optimality to Efficiently and Privately Retrieve Security Updates. In *Financial Cryptography and Data Security: 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers* (Berlin, Heidelberg, 2013), A.-R. Sadeghi, Ed., Springer Berlin Heidelberg, pp. 386–394.

[14] CAPPOS, J., BAKER, S., PLICHTA, J., NYUGEN, D., HARDIES, J., BORGARD, M., JOHNSTON, J., AND HARTMAN, J. H. Stork: package management for distributed VM environments. In *The 21st Large Installation System Administration Conference, LISA'07* (2007).

[15] CAPPOS, J., SAMUEL, J., BAKER, S., AND HARTMAN, J. H. A look in the mirror: Attacks on package managers. In *Proceedings of the 15th ACM conference on Computer and communications security* (2008), ACM, pp. 565–574.

[16] CAPPOS, J., SAMUEL, J., BAKER, S., AND HARTMAN, J. H. Package management security. Tech. Rep. TR-08-02, University of Arizona, 2008.

[17] CAPPPOS, J. *Stork: Secure Package Management for VM Environments*. Dissertation, University of Arizona, 2008.

[18] CASTRO, M., AND LISKOV, B. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 1999), OSDI '99, USENIX Association, pp. 173–186.

[19] CATTANEO, G., CATUOGNO, L., SORBO, A. D., AND PERSIANO, P. The Design and Implementation of a Transparent Cryptographic File System for UNIX. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2001), USENIX Association, pp. 199–212.

[20] CHUN, B.-G., MANIATIS, P., SHENKER, S., AND KUBIATOWICZ, J. Attested Append-only Memory: Making Adversaries Stick to Their Word. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 189–204.

[21] CORBET, J. An attempt to backdoor the kernel. http://lwn.net/Articles/57135/, 2003.

[22] CORBET, J. The cracking of kernel.org. http://www.linuxfoundation.org/news-media/blogs/browse/2011/08/cracking-kernelorg, 2011.

[23] DEBIAN. Debian Investigation Report after Server Compromises. https://www.debian.org/News/2003/20031202, 2003.

[24] DEBIAN. Security breach on the Debian wiki 2012-07-25. https://wiki.debian.org/DebianWiki/SecurityIncident2012, 2012.

[25] DOCKER INC. Docker Hub. https://hub.docker.com/.

[26] DOUCEUR, J. R., AND WATTENHOFER, R. P. Optimizing file availability in a secure serverless distributed file system. In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems* (2001), pp. 4–13.

[27] DUCHAMP, D. A Toolkit Approach to Partially Connected Operation. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 1997), USENIX Association, pp. 23–23.

[28] FRIELDS, P. W. Infrastructure report, 2008-08-22 UTC 1200. https://www.redhat.com/archives/fedora-announce-list/2008-August/msg00012.html, 2008.

[29] FU, K., KAASHOEK, M. F., AND MAZIÈRES, D. Fast and Secure Distributed Read-only File System. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4* (Berkeley, CA, USA, 2000), OSDI'00, USENIX Association.

[30] FU, K. E. Group sharing and random access in cryptographic storage file systems. Master's thesis, Massachusetts Institute of Technology, 1999.

[31] GENTOO LINUX. rsync.gentoo.org: rotation server compromised. https://security.gentoo.org/glsa/200312-01, 2003.

[32] GESBERT, L., AND MEHNERT, H. Signing the OPAM repository. https://opam.ocaml.org/blog/Signing-the-opam-repository/, 2015.

[33] GITHUB, INC. Public Key Security Vulnerability and Mitigation. https://github.com/blog/1068-public-key-security-vulnerability-and-mitigation, 2012.

[34] GNU SAVANNAH. Compromise2010. https://savannah.gnu.org/maintenance/Compromise2010/, 2010.

[35] GOH, E.-J., SHACHAM, H., MODADUGU, N., AND BONEH, D. SiRiUS: Securing Remote Untrusted Storage. In *The 10th Annual Network and Distributed System Security Symposium* (San Diego, CA, Feb. 2003), The Internet Society, pp. 131–145.

[36] GOODIN, D. Attackers sign malware using crypto certificate stolen from Opera Software. http://arstechnica.com/security/2013/06/attackers-sign-malware-using-crypto-certificate-stolen-from-opera-software/, 2013.

[37] HAEBERLEN, A., KOUZNETSOV, P., AND DRUSCHEL, P. PeerReview: Practical Accountability for Distributed Systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 175–188.

[38] HO, C., VAN RENESSE, R., BICKFORD, M., AND DOLEV, D. Nysiad: Practical Protocol Transformation to Tolerate Byzantine Failures. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2008), NSDI'08, USENIX Association, pp. 175–188.

[39] KALLAHALLA, M., RIEDEL, E., SWAMINATHAN, R., WANG, Q., AND FU, K. Plutus: Scalable Secure File Sharing on Untrusted Storage. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2003), FAST '03, USENIX Association, pp. 29–42.

[40] KEYBASE. Introducing the Keybase filesystem, 2016. https://keybase.io/docs/kbfs.

[41] KORN, D., MACDONALD, J., MOGUL, J., AND VO, K. The VCDIFF Generic Differencing and Compression Data Format. RFC 3284 (Proposed Standard). https://tools.ietf.org/html/rfc3284, June 2002.

[42] KORN, D. G., AND VO, K.-P. A Generic Differencing and Compression Data Format. Tech. Rep. HA1630000-021899-02TM, AT&T Labs - Research, February, 1999.

[43] KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., AND WONG, E. Zyzzyva: Speculative Byzantine Fault Tolerance. In *Proceedings of Twenty-first*

ACM SIGOPS Symposium on Operating Systems Principles (New York, NY, USA, 2007), SOSP '07, ACM, pp. 45–58.

[44] KUHN, B. M. News: IMPORTANT: Information Regarding Savannah Restoration for All Users. https://savannah.gnu.org/forum/forum.php?forum_id=2752, 2003.

[45] KUPPUSAMY, T. K., DIAZ, V., STUFFT, D., AND CAPPOS, J. PEP 458 – Securing the Link from PyPI to the End User. https://www.python.org/dev/peps/pep-0458/, 2013.

[46] KUPPUSAMY, T. K., DIAZ, V., STUFFT, D., AND CAPPOS, J. PEP 480 – Surviving a Compromise of PyPI. https://www.python.org/dev/peps/pep-0480/, 2014.

[47] KUPPUSAMY, T. K., TORRES-ARIAS, S., DIAZ, V., AND CAPPOS, J. Diplomat: Using Delegations to Protect Community Repositories. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, Mar. 2016), USENIX Association, pp. 567–581.

[48] LAMPORT, L., SHOSTAK, R., AND PEASE, M. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst. 4*, 3 (July 1982), 382–401.

[49] LEAP ENCRYPTION ACCESS PROJECT. New releases for a new year - LEAP. https://leap.se/en/2014/darkest-night, 2014.

[50] LEVIN, D., DOUCEUR, J. R., LORCH, J. R., AND MOSCIBRODA, T. TrInc: Small Trusted Hardware for Large Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2009), NSDI'09, USENIX Association, pp. 1–14.

[51] LI, J., KROHN, M., MAZIÈRES, D., AND SHASHA, D. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 9–9.

[52] LI, J., AND MAZIÉRES, D. Beyond One-third Faulty Replicas in Byzantine Fault Tolerant Systems. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2007), NSDI'07, USENIX Association.

[53] LI, J., REIHER, P., AND POPEK, G. J. Resilient self-organizing overlay networks for security update delivery. *Selected Areas in Communications, IEEE Journal on 22*, 1 (2004), 189–202.

[54] MAGNUSSON, H. The PHP project and Code Review. http://bjori.blogspot.com/2010/12/php-project-and-code-review.html, 2010.

[55] MAHAJAN, P., SETTY, S., LEE, S., CLEMENT, A., ALVISI, L., DAHLIN, M., AND WALFISH, M. Depot: Cloud Storage with Minimal Trust. *ACM Trans. Comput. Syst. 29*, 4 (Dec. 2011), 12:1–12:38.

[56] MAHESHWARI, U., VINGRALEK, R., AND SHAPIRO, W. How to Build a Trusted Database System on Untrusted Storage. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4* (Berkeley, CA, USA, 2000), OSDI'00, USENIX Association.

[57] MAZIÈRES, D., AND SHASHA, D. Building Secure File Systems out of Byzantine Storage. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing* (New York, NY, USA, 2002), PODC '02, ACM, pp. 108–117.

[58] MERKLE, R. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology — CRYPTO '87*, C. Pomerance, Ed., vol. 293 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1988, pp. 369–378.

[59] MICROSOFT, INC. Flame malware collision attack explained. http://blogs.technet.com/b/srd/archive/2012/06/06/more-information-about-the-digital-certificates-used-to-sign-the-flame-malware.aspx, 2012.

[60] MILLER, E., LONG, D., FREEMAN, W., AND REED, B. Strong security for distributed file systems. In *Performance, Computing, and Communications, 2001. IEEE International Conference on.* (2001), IEEE, pp. 34–40.

[61] MOGUL, J., KRISHNAMURTHY, B., DOUGLIS, F., FELDMANN, A., GOLAND, Y., VAN HOFF, A., AND HELLERSTEIN, D. Delta encoding in HTTP. RFC 3229 (Proposed Standard). https://tools.ietf.org/html/rfc3229, Jan. 2002.

[62] MOGUL, J. C., DOUGLIS, F., FELDMANN, A., AND KRISHNAMURTHY, B. Potential Benefits of Delta Encoding and Data Compression for HTTP. In *Proceedings of the ACM SIGCOMM '97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 1997), SIGCOMM '97, ACM, pp. 181–194.

[63] MULLENWEG, M. Passwords Reset. https://wordpress.org/news/2011/06/passwords-reset/, 2011.

[64] MÓNICA, D., AND DOCKER, INC. Introducing Docker Content Trust. https://blog.docker.com/2015/08/content-trust-docker-1-8/, 2015.

[65] OPREA, A., AND REITER, M. K. Integrity Checking in Cryptographic File Systems with Constant Trusted Storage. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium* (Berkeley, CA, USA, 2007), SS'07, USENIX Association, pp. 13:1–13:16.

[66] PHILIPS, B. Evaluate The Update Framework. https://github.com/appc/spec/issues/211, 2015.

[67] POPA, R. A., LORCH, J. R., MOLNAR, D., WANG, H. J., AND ZHUANG, L. Enabling Security in Cloud Storage SLAs with CloudProof. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2011), USENIXATC'11, USENIX Association.

[68] PRIME DIRECTIVE, INC. Development - Flynn. https://flynn.io/docs/development, 2015.

[69] PYTHON SOFTWARE FOUNDATION. PyPI - the Python Package Index: Python Package Index. https://pypi.python.org/pypi.

[70] RED HAT, INC. Infrastructure report, 2008-08-22 UTC 1200. https://rhn.redhat.com/errata/RHSA-2008-0855.html, 2008.

[71] RUBYGEMS.ORG. RubyGems.org — your community gem host. https://rubygems.org/.

[72] RUBYGEMS.ORG. Data Verification. http://blog.rubygems.org/2013/01/31/data-verification.html, 2013.

[73] SAMUEL, J., MATHEWSON, N., CAPPOS, J., AND DINGLEDINE, R. Survivable key compromise in software update systems. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), ACM, pp. 61–72.

[74] SCHNEIDER, F. B. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv. 22*, 4 (Dec. 1990), 299–319.

[75] SHAY, X., AND SQUARE, INC. Securing RubyGems with TUF, Part 1. https://goo.gl/XO4AHu, 2013.

[76] SHAY, X., AND SQUARE, INC. Securing RubyGems with TUF, Part 2. https://goo.gl/yymo8z, 2013.

[77] SHAY, X., AND SQUARE, INC. Securing RubyGems with TUF, Part 3. https://goo.gl/pPKcgB, 2013.

[78] SHRAER, A., CACHIN, C., CIDON, A., KEIDAR, I., MICHALEVSKY, Y., AND SHAKET, D. Venus: Verification for Untrusted Cloud Storage. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop* (New York, NY, USA, 2010), CCSW '10, ACM, pp. 19–30.

[79] SINGH, A., FONSECA, P., KUZNETSOV, P., RODRIGUES, R., AND MANIATIS, P. Zeno: Eventually Consistent Byzantine-fault Tolerance. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2009), NSDI'09, USENIX Association, pp. 169–184.

[80] SLASHDOT MEDIA. About. http://sourceforge.net/about.

[81] SLASHDOT MEDIA. phpMyAdmin corrupted copy on Korean mirror server. https://sourceforge.net/blog/phpmyadmin-back-door/, 2012.

[82] SMITH, J. K. Security incident on Fedora infrastructure on 23 Jan 2011. https://lists.fedoraproject.org/pipermail/announce/2011-January/002911.html, 2011.

[83] STEFANOV, E., VAN DIJK, M., JUELS, A., AND OPREA, A. Iris: A Scalable Cloud File System with Efficient Integrity Checks. In *Proceedings of the 28th Annual Computer Security Applications Conference* (New York, NY, USA, 2012), ACSAC '12, ACM, pp. 229–238.

[84] STEIN, C. A., HOWARD, J. H., AND SELTZER, M. I. Unifying File System Protection. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2001), USENIX Association, pp. 79–90.

[85] TECTONIC/COREOS, INC. Tectonic with Distributed Trusted Computing, 2016. https://tectonic.com/trusted-computing/.

[86] THE FREEBSD PROJECT. FreeBSD.org intrusion announced November 17th 2012. http://www.freebsd.org/news/2012-compromise.html, 2012.

[87] THE PHP GROUP. php.net security notice. http://www.php.net/archive/2011.php#id2011-03-19-1, 2011.

[88] THE PHP GROUP. A further update on php.net. http://php.net/archive/2013.php#id2013-10-24-2, 2013.

[89] THE UPDATE FRAMEWORK. Developer Tools. https://github.com/theupdateframework/tuf/blob/develop/tuf/README-developer-tools.md, 2014.

[90] THE UPDATE FRAMEWORK. Repository Management. https://github.com/theupdateframework/tuf/blob/develop/tuf/README.md, 2014.

[91] TRUSTED COMPUTING GROUP. Trusted Platform Module (TPM), 2016. https://www.trustedcomputinggroup.org/work-groups/trusted-platform-module/.

[92] VAN DIJK, M., SARMENTA, L. F., O'DONNELL, C. W., AND DEVADAS, S. Proof of freshness: How to efficiently use an online single secure clock to secure shared untrusted memory. Tech. Rep. CSG Memo 496, Massachusetts Institute of Technology, 2006.

[93] VOSS, L. Newly Paranoid Maintainers. http://blog.npmjs.org/post/80277229932/newly-paranoid-maintainers, 2014.

[94] WELL-TYPED LLP. Improving Hackage security. http://www.well-typed.com/blog/2015/04/improving-hackage-security/, 2015.

[95] YUMEREFENDI, A. R., AND CHASE, J. S. Strong Accountability for Network Storage. *Trans. Storage 3*, 3 (Oct. 2007).

CAB-FUZZ: Practical Concolic Testing Techniques for COTS Operating Systems

Su Yong Kim*§ Sangho Lee† Insu Yun† Wen Xu†
Byoungyoung Lee¶ Youngtae Yun* Taesoo Kim†

*The Affiliated Institute of ETRI †Georgia Institute of Technology ¶Purdue University

Abstract

Discovering the security vulnerabilities of commercial off-the-shelf (COTS) operating systems (OSes) is challenging because they not only are huge and complex, but also lack detailed debug information. Concolic testing, which generates all feasible inputs of a program by using symbolic execution and tests the program with the generated inputs, is one of the most promising approaches to solve this problem. Unfortunately, the state-of-the-art concolic testing tools do not scale well for testing COTS OSes because of state explosion. Indeed, they often fail to find a single bug (or crash) in COTS OSes despite their long execution time.

In this paper, we propose CAB-FUZZ (Context-Aware and Boundary-focused), a *practical* concolic testing tool to quickly explore interesting paths that are highly likely triggering real bugs without debug information. First, CAB-FUZZ prioritizes the boundary states of arrays and loops, inspired by the fact that many vulnerabilities originate from a lack of proper boundary checks. Second, CAB-FUZZ exploits real programs interacting with COTS OSes to construct proper contexts to explore deep and complex kernel states without debug information. We applied CAB-FUZZ to Windows 7 and Windows Server 2008 and found *21 undisclosed unique crashes*, including two local privilege escalation vulnerabilities (CVE-2015-6098 and CVE-2016-0040) and one information disclosure vulnerability in a cryptography driver (CVE-2016-7219). CAB-FUZZ found vulnerabilities that are non-trivial to discover; five vulnerabilities have existed for 14 years, and we could trigger them even in the initial version of Windows XP (August 2001).

1 Introduction

Concolic testing is a well-known approach to *automatically* detect software vulnerabilities [8]. Empowered by its symbolic interpretation of the input, it generates and explores all feasible states in a program and thoroughly checks whether a certain security property can be violated. In particular, it has shown its effectiveness for *small* applications and/or applications with *source code*. For example, Avgerinos et al. [1] found more than 10,000 bugs in about 4,000 small applications. Also, Ramos and Engler [40] found 67 bugs in various open-source projects, such as BIND, OpenSSL, and the Linux kernel.

However, concolic testing *does not scale well for complex and large software* [5, 8, 13, 48], such as commercial off-the-shelf (COTS) operating systems (OSes). The complete concolic execution of COTS OSes would never terminate in a reasonable amount of time due to the well-known limitation of the symbolic execution, *state (or path) explosion*, where the number of feasible program states increases exponentially (e.g., once reaching a loop statement). Since the COTS OSes have massive implementation complexity, testing using symbolic execution ends up exploring a very small portion of program states, i.e., it cannot test deep execution paths.

Moreover, a proprietary nature of COTS OSes prevents concolic testing from exploring program states with *pre-contexts*. Unlike the open-source kernel for which the internal documentation and all test suites are publicly available [41, 50], COTS OSes do not provide such comprehensive information. Although manual annotation on the interface can help increase code coverage and detect logical bugs [27], it also does not scale. For these reasons, concolic execution on COTS OSes cannot explore program states that are only reachable after undergoing complex runtime operations.

In this paper, we propose CAB-FUZZ (Context-Aware and Boundary-focused), a practical system specialized to detect vulnerabilities in COTS OSes based on concolic testing. First, to overcome the scalability limitation of concolic testing, CAB-FUZZ *prioritizes* states likely having vulnerabilities. This prioritization is based on the observation that a majority of critical security bugs (e.g., memory corruption and information disclosure) originate

§This work is done while this author was a visiting scholar in Georgia Institute of Technology.

from *a lack of proper boundary checks*. This is why compilers and even hardware have adopted boundary-check mechanisms, such as SoftBound [37], SafeStack [28], and Intel Memory Protection Extensions (MPX) [20]. Therefore, we instruct CAB-FUZZ to generate and explore the boundary states of arrays and loops first, thereby detecting vulnerabilities as early as possible before exploding in terms of program states.

Second, to construct pre-contexts of COTS OSes without detailed debug information, CAB-FUZZ refers to *real programs* as a concolic-execution template. Since such a program frequently interacts with the COTS OSes to perform a certain operation, it embodies sufficient information and logic that constructs pre-contexts for using OS functions. Thus, if CAB-FUZZ runs a real program until it calls any target OS function that we are interested in, CAB-FUZZ is able to prepare with proper pre-contexts to initiate concolic testing correctly.

We implemented CAB-FUZZ based on a popular concolic testing tool, S2E [10], and evaluated it with two popular COTS OSes, Windows 7 and Windows Server 2008, especially for the 274 device drivers shipped with them. Since our approaches are general and independent of the OS, we believe they can be applied to currently unsupported OSes in the future.

In total CAB-FUZZ discovered *21 unique crashes* of six device drivers developed by Microsoft and ESET (§5). Among them we reported six reproducible crashes to Microsoft and one reproducible crash to ESET. Microsoft confirmed that three of them were undisclosed vulnerabilities and could be abused by a guest account for *local privilege escalation* (CVE-2015-6098 and CVE-2016-0040) and *information disclosure in a cryptography driver* (CVE-2016-7219). Especially, the later vulnerability even existed in the latest versions of Windows (Windows 10 and Windows Server 2016). Microsoft acknowledged the other three reports demanding administrator privilege and ESET fixed the bug we reported.

This evaluation result arguably demonstrates the effectiveness of CAB-FUZZ in finding vulnerabilities in COTS OSes despite its lack of completeness. CAB-FUZZ may not be able to trigger sophisticated bugs unrelated to boundary states. However, because of the fundamental scalability limitation of concolic testing, complete concolic testing is infeasible especially for large software. One of the contributions of CAB-FUZZ is that it changes the way we think of concolic testing—sacrificing completeness in a degree—to make it practical. Microsoft invests huge engineering efforts and computational resources in finding vulnerabilities, but CAB-FUZZ still discovered many different vulnerabilities in the Windows kernel using relatively moderate engineering efforts and computational resources. Specifically, we want to emphasize that Microsoft made fuzzing mandatory for every untrusted interface for every product, and their fuzzing solution has been running 24/7 since 2007 for a total of over 500 machine years [3]. However, despite this effort, CAB-FUZZ was able to discover 14-year-old bugs in Windows' kernel device drivers (§5.3).

This paper makes the following contributions.

- **Practical Techniques.** CAB-FUZZ makes concolic testing *practical* by addressing its two important challenges: state explosion and missing execution contexts. CAB-FUZZ prioritizes boundary conditions to trigger a crash before explosion and refers to a real application to construct proper execution contexts.
- **Evaluation and In-depth Analysis.** We analyzed the implementation of COTS OSes in detail to figure out why CAB-FUZZ was able to detect their vulnerabilities effectively compared to conventional techniques.
- **Real-world Impact.** CAB-FUZZ discovered *21 unique crashes* of device drivers for Windows 7 and Windows Server 2008. We reported all reproducible crashes to the vendors. They confirmed that four of the reported crashes were critical and fixed them. Specifically, two of them were privilege escalation vulnerabilities and one was an information disclosure vulnerability in a cryptography driver.

The rest of this paper is organized as follows. §2 describes the challenges of performing concolic testing for COTS OSes. §3 depicts CAB-FUZZ and §4 describes its implementation. §5 evaluates CAB-FUZZ's vulnerability-finding effectiveness. §6 discusses the various aspects of CAB-FUZZ including its limitations, and §7 presents related work. §8 concludes the paper.

2 Challenges for COTS OSes

This section elaborates on the challenges involved in performing concolic testing for COTS OSes to clearly motivate our proposed system, CAB-FUZZ.

2.1 Binary

Automated binary analysis is necessary for production software (e.g., COTS OS) because (1) it usually contains third-party binaries and libraries without source code, (2) its behavior can be changed due to compiler optimization or linking, and (3) its code can be written with multiple programming languages, making source code analysis difficult. However, the following two challenges make concolic testing for COTS OSes unpractical.

Missing Documentation and Test-suites. When doing automated testing, especially for COTS OSes, a lack of source code and document is a critical hurdle because most of the communication interfaces between user- and kernel-space are undocumented (often intentionally) and vary dramatically across versions [21]. Further, COTS

```
1  // global arrays
2  bool flag_table[125];
3  void (*fn_table[36])(); //function pointer array
4
5  int dispatch_device_io_control(unsigned long ctrl_code,
6                                 unsigned long *buf) {
7    switch (ctrl_code) {
8    case 0x8fff23cc:
9    case 0x8fff23c8:
10     // sanitizing conditions (simplified)
11     if (buf[0] > 246 || buf[1] > 124 || buf[2] > 36)
12         return -1;
13
14     if (flag_table[buf[1]]) {
15       // buf[2] == 36 -> out-of-bound access
16       (*fn_table[buf[2]])();
17     }
18
19     for (int i = 1; i <= buf[0]; ++i) { ... }
20
21   // NOTE. the below included to comprehensively illustrate
22   // the effectiveness of on-the-fly technique.
23   // Not exist in the original NDProxy
24   case 0x8fff23c4:
25     // set all elements of flag_table to true
26     for (int i = 0; i < 125; i++)
27       flag_table[i] = true;
28     ...
29   }
30 }
```

Figure 1: A simplified code snippet reconstructed from NDProxy vulnerability (CVE-2013-5065) [11]. It resulted in a local privilege escalation in Windows XP and Server 2003.

OSes often do not provide test suites such that it is difficult to generate proper input values that pass input validation routines at an early state. This prevents the concolic testing procedure from reaching later and deeper stages. Even the state-of-the-art techniques (S2E [10] and Dowser [19]) rely on unit tests to pass input validation routines.

Handling Symbolic Memory. There are two common ways to handle symbolic memory in concolic testing: treating it as a symbolic array (*symbolization*) or concretizing it (*concretization*). Memory symbolization is typically used to avoid the state explosion problem because it efficiently abstracts the execution state. However, memory symbolization is not suitable for a COTS binary because it heavily uses the static information (e.g., object size) for performance optimization, which is often unavailable. Further, it produces complex constraints that are barely solvable in large-scale, real-world software.

Therefore, CAB-FUZZ concretizes every symbolic memory as it produces solvable constraints even for large-scale software. But, it has to cope with the state explosion problem as we discuss in the next section.

2.2 State Explosion

We illustrate state explosion with an NDProxy vulnerability (CVE-2013-5065) and S2E [10].

CVE-2013-5065. Figure 1 shows a simplified code snippet reverse-engineered from the NDProxy kernel driver. The `dispatch_device_io_control` function handles the requests of a user-mode process. `ctrl_code` and `buf` are inputs from a user-mode process, where `ctrl_code` represents an operation and `buf` contains user data.

According to our analysis, this vulnerability originated from the misverification of `buf[2]` at Line 11. `buf[2]` is used as an index to refer to `fn_table` and it should lie between `0` and `35` to avoid memory access violations. In principle, having `ctrl_code` and `buf` as symbolic variables, S2E [10] is supposed to identify the offending input satisfying the vulnerable condition. However, we found it suffers from state explosion.

State Explosion Problem. We carefully adjusted S2E to check the code (Figure 1) as a preliminary experiment (§5.1). Due to state explosion, it took *two hours* while consuming up to 15 GB of memory to detect the vulnerability. First, S2E explored all feasible paths of *symbolic memory*—a memory region a symbolic variable controls. The code had at least two symbolic memory arrays: `fn_table` and `flag_table`, where `fn_table` generated 37 states due to the condition of `buf[2]` at Line 11, and `flag_table` generated 125 states due to the condition of `buf[1]` at Line 11. Second, S2E explored all possible paths of a *loop* controlled by a symbolic variable. This code had a loop controlled by `buf[0]` at Line 19, generating at least 247 states in our observation. In total, S2E generated more than a million states just for two symbolic memories and a single loop.

Exploring all feasible paths of a program is difficult in practice due to state explosion. Instead, CAB-FUZZ prioritizes interesting paths that more likely trigger vulnerabilities. For example, the vulnerability in Figure 1 is triggered when `buf[2]` has the upper-bound value `36`. Focusing on such *boundary states* allows us to detect many vulnerabilities while avoiding state explosion (§3.2).

2.3 Missing Execution Contexts

To avoid state explosion, concolic testing tools need to check individual functions instead of the entire program from the beginning. However, functions can have close relationships with each other such that we cannot establish proper contexts when skipping some of them (e.g., a function for initializing shared variables) [40].

Figure 1 also shows a crash example that context-unaware concolic testing tools cannot detect (Lines 14, 16, and 27). In fact, `fn_table[buf[2]]` will be executed only after `dispatch_device_io_control` with `0x8fff23c4` as `ctrl_code` has been called first since it depends on a global array `flag_table`. When testing such a function, existing concolic testing tools just treat its input parameters as symbolic variables, ignoring context such as the sequence of function calls. However, this cannot generate a crash because no elements of `flag_table` have the value required for the crash. Therefore, existing tools cannot detect the bug in our example.

CAB-FUZZ targets COTS OSes such that it aims to solve this problem without relying on any prior knowl-

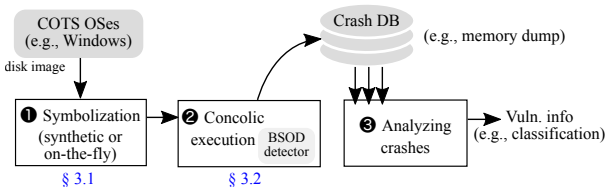


Figure 2: An overview of CAB-FUZZ's workflow.

edge (e.g., annotation). Our basic idea is to run a *real program*, instead of a synthetic program, to let it construct pre-contexts. Later, when the program is calling a target function, CAB-FUZZ initiates concolic testing *on-the-fly*. This allows us to get enough pre-contexts to test the target function with minimal overhead (details are in §3.1.2.)

3 Design

In this section, we describe in detail CAB-FUZZ's design and the techniques that allow for concolic execution for COTS OSes. CAB-FUZZ is a full-fledged vulnerability-detection system for COTS binaries, and in particular, it aims to make concolic testing (see §2) practical in the context of COTS OSes.

Figure 2 depicts an overview of CAB-FUZZ. First, it takes a disk image of the targeted COTS OS as an input. Then, it determines when to start symbolic execution either by synthetic symbolization (§3.1.1) or on-the-fly symbolization (§3.1.2). After deciding what to symbolize, CAB-FUZZ performs the concolic testing. In order to address the state explosion problem, CAB-FUZZ employs two new techniques, namely, array-boundary prioritization (§3.2.1) and loop-boundary prioritization (§3.2.2), which focus on boundary states (§3.2). Once CAB-FUZZ observes a kernel crash during the symbolic execution, it attempts to generate concrete input and a crash report to help reproduce the observed crash.

3.1 Symbolization for Kernel

The goal of CAB-FUZZ is to detect the vulnerabilities in the kernel using concolic testing. In particular, CAB-FUZZ symbolizes a certain memory location during kernel execution such that any instruction involving this location is symbolically executed. Although this procedure resembles generic concolic testing methods, we specialize CAB-FUZZ for handling COTS OSes by considering two important issues: *when to start the symbolic execution* and *what memory regions to symbolize*. The kernel can be considered as a long-running process or system service, and the majority of its functional components depend on previous kernel execution states. CAB-FUZZ takes two different approaches in this regard: synthetic symbolization (§3.1.1) and on-the-fly symbolization (§3.1.2). Overall, synthetic symbolization launches a previously built user-space program and explicitly starts the symbolic execution phase. On the other hand, on-the-fly symbolization retrofits the existing user-space programs to better construct the legitimate kernel execution contexts and seamlessly starts the symbolic execution at a certain execution point.

3.1.1 Synthetic Symbolization

Synthetic symbolization launches a previously built user-space program that initiates the symbolic execution. This largely follows previous concolic execution techniques in that CAB-FUZZ also launches synthetic programs to start the symbolic execution phase. The key difference is that CAB-FUZZ tailors the user-space programs to test kernel device drivers. Our synthetic program invokes a function controlling an IO device (i.e., `NtDeviceIoControlFile`) while symbolizing its parameters.

Figure 3 shows example code to test `NtDeviceIoControlFile`. In particular, for each device driver, we obtain the corresponding device driver handle at Line 17. Using this handle, CAB-FUZZ invokes `NtDeviceIoControlFile` while symbolizing the two parameters, `ctrl_code` and `in_buf`, which primarily control the behavior of a device driver (see Figure 1). We observed that symbolizing the size of `in_buf` resulted in state explosion, leading us to decide not to symbolize it (explained later). The memory symbolization is carried out by utilizing existing runtime helper functions in the concolic execution engine (i.e., `s2e_make_symbolic`). Once these two parameters are symbolized, CAB-FUZZ symbolically interprets these parameters while executing `NtDeviceIoControlFile`.

State Explosion due to Input Buffer Size Symbolization. We explain why input buffer size symbolization generates state explosion. Windows provides three methods to deliver a user-space input buffer to the kernel, configured using the lowest two bits of `ctrl_code` [33]. The first method, *buffered I/O*, allocates a kernel memory buffer whose size is the same as that of a user input buffer and copies the input buffer's content to the kernel buffer. The buffered I/O, however, generates state explosion, as shown in Figure 4. At Line 9, `in_buf_size` is used as a condition of the for loop, so it generates `0x7FFF0000` states even with the constraint at Line 7.

The other two methods (*direct I/O* and *neither buffered nor direct I/O*) do not directly generate state explosion since they let a kernel device driver access the user buffer via a memory descriptor list (MDL) or virtual address. However, since we focus on COTS OSes, we do not know which method a target driver uses to access a user input buffer. Consequently, CAB-FUZZ should symbolize the for loop no matter which method the target driver uses.

3.1.2 On-the-Fly Symbolization

As shown in §2.3, existing concolic testing tools cannot check individual target functions due to the lack of context awareness. To this end, on-the-fly symbolization retrofits the real user-space programs to better construct

```
HANDLE device_handle;
unsigned long in_buf[BUF_SIZE] = {0};
unsigned long out_buf[BUF_SIZE] = {0};
unsigned long ctrl_code = 0;
NTSTATUS status;
UNICODE_STRING device_name;
OBJECT_ATTRIBUTES object_attributes;
ACCESS_MASK max_allowed_access;
IO_STATUS_BLOCK io_status_block;

// get maximum access allowed for the target device driver
max_allowed_access = get_allowed_access(&device_name);

obect_attributes.ObjectName = &device_name;

// get handle of the target device driver
status = NtCreateFile(&device_handle, max_allowed_access,
                      &object_attributes, ...);

if (status)
  return -1; //cannot get a handle

// initate concolic execution and symoblize params
cab_start_concolic_testing();

s2e_make_symbolic(&ctrl_code, sizeof(ctrl_code), "code");
s2e_make_symbolic(&in_buf, sizeof(in_buf), "buf");

// targeted call
NtDeviceIoControlFile(
  device_handle,    // handle to target device
  NULL,             // A handle to an event
  NULL,             // ApcRoutine procedure
  NULL,             // a pointer to pass to ApcRoutine
  &io_status_block, // receive the final completion status
  ctrl_code,        // a control function to be executed
  &in_buf,          // input buffer
  BUF_SIZE,         // input buffer size
  &out_buf,         // output buffer
  BUF_SIZE);        // output buffer size

// terminate and generate a testcase
s2e_kill_state(0, "Successfully done");
```

Figure 3: Example code of the synthetic symbolization testing the `NtDeviceIoControlFile` function (see §3.1.1 for explanation). `get_allowed_access()` is related to the access permission per the driver (see §4.1 for more details). The prototype of `NtDeviceIoControlFile` function can be found in [35].

the legitimate kernel execution contexts and seamlessly starts the symbolic execution at a certain execution point. Specifically, unlike existing concolic testing tools, our on-the-fly concolic testing tries to satisfy the *pre-contexts* of a function to crash in our best effort by following the real execution procedure of a COTS binary, as shown in Figure 5. It (1) runs and monitors the execution of a user-space program, (2) lets the program and kernel construct pre-contexts, (3) monitors input values to a target function and selects some of them, and (4) performs runtime concolic testing while designating the selected values as symbolic variables.

For example, to test `dispatch_device_io_control` in Figure 1, our tool runs a user-space program containing the code, ensuring the initialization has been called (i.e., `ctrl_code = 0x8fff23c4`) before other `NtDeviceIoControlFile` calls. Since the pre-context is now fulfilled, concolic testing can automatically generate the value that results in a crash.

```
// ctrl_code, in_buf_size, and in_buf are given from
// a user-space process. kernel_mem is a kernel-space buffer

#define USER_ADDR_MAX 0x7fff0000

if (ctrl_code & 3 == 0) { // Buffered I/O
    if (in_buf_size < USER_ADDR_MAX) {
        ...
        for (int i = 0; i < in_buf_size; i++) {
            kernel_mem[i] = in_buf[i];
        }
        ...
    }
}
```

Figure 4: Pseudo code showing why symbolizing an input buffer size generates state explosion during concolic testing.

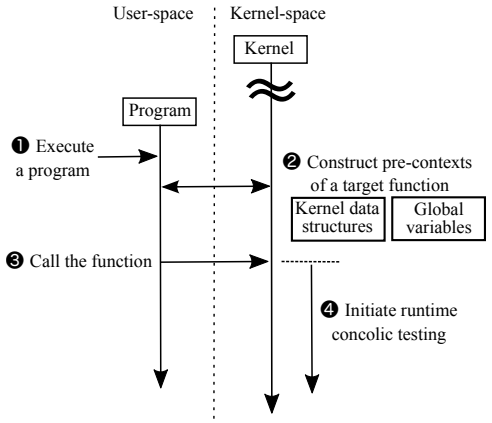


Figure 5: Overall procedures of on-the-fly concolic testing: ❶ CAB-FUZZ executes a real user-space program; ❷ CAB-FUZZ lets the program and kernel interact with each other to construct the pre-contexts of a target function; ❸ the program calls a target function; ❹ CAB-FUZZ hooks the event and initiates runtime concolic testing from this point.

Furthermore, our on-the-fly concolic testing method can work with COTS binaries that provide only partial information. Many COTS binaries lack full documentation, so we cannot obtain all the information to test target functions. This makes existing concolic testing tools ineffective in practice because it is difficult to pass the sanitization routines without satisfying basic conditions among inputs. Even in such a case, our on-the-fly symbolization has a chance to bypass uninteresting sanitization routines, yet effectively test the target function by deriving input conditions from a real execution [46].

3.2 Boundary-state Prioritization

In this section, we introduce *boundary-state prioritization* that attempts to overcome the state explosion due to symbolic arrays and loops in COTS OSes. The key idea of the boundary-state prioritization is to defer the analysis of uninteresting states based on the likelihood of security vulnerability (e.g., memory corruption and disclosure). In other words, we focus on triggering security vulnerabilities via concolic execution while compromising the completeness of testing for performance and scalability.

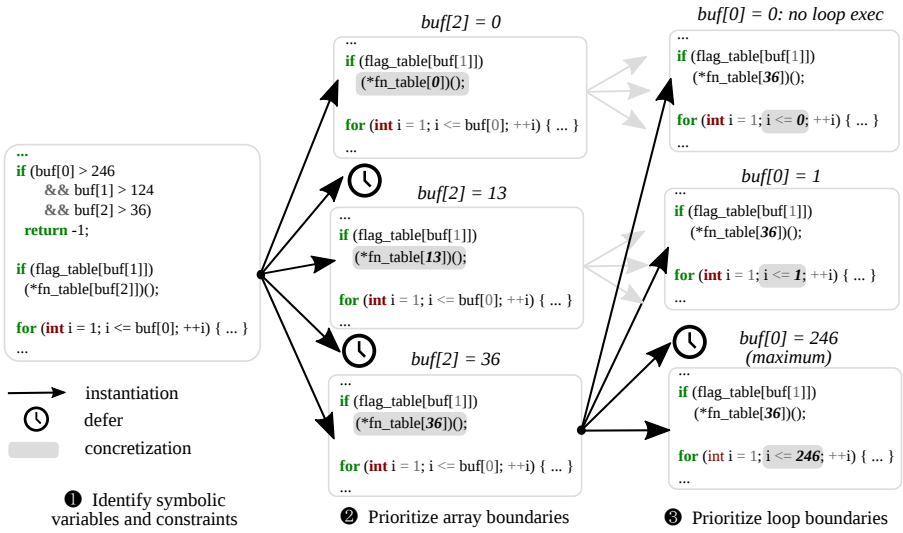


Figure 6: Overall procedures of boundary-state prioritization: ❶ CAB-FUZZ identifies symbolic variables and constraints; ❷ CAB-FUZZ selectively concretizes symbolic memories according to their boundary information; ❸ CAB-FUZZ selectively concretizes loops using their boundary information (in this example, we do not symbolized `flag_table` for simplicity.)

Figure 6 shows the overall procedures. First, it figures out constraints that limit the range of symbolic variables using KLEE's range analysis function [6]. Second, it detects symbolic memories controlled by the symbolic variables and selectively concretizes them according to their boundary information (*array-boundary prioritization*). Third, it detects loops and selectively iterates through them using the boundary information (*loop-boundary prioritization*). Without our prioritization techniques, the total number of states exponentially increases according to the number of symbolic memories and loops. If the number of symbolic arrays and loops is n and the number of possible states of each symbolic array or loop is s_i, the total number of states to explore will be $\prod_{i=1}^{n} s_i$. In contrast, our techniques test $\prod_{i=1}^{n} c = c^n$ states first, where c is a constant.

3.2.1 Array-boundary Prioritization

We explain our array-boundary prioritization technique with two symbolic memories `flag_table` and `fn_table` in Figure 1 and Figure 6. As we discussed in §2.2, `flag_table` generates 125 states and `fn_table` generates 37 states, which result in $125 \times 37 = 4,625$ states total.

Exploring all states is challenging, especially when the length of a target array is long and/or many symbolic memories and loops are associated with it. Instead, CAB-FUZZ drives symbolic execution to visit boundary cases first, which highly likely have problems. Specifically, CAB-FUZZ creates two states for each symbolic memory by solving the associated constraints: the *lowest* memory address and the *highest* memory address. Note that the two boundary states could result in exceptions due to crashes or boundary checks. To proceed the test, CAB-FUZZ additionally creates a state for an arbitrary memory address between them.

The second step of Figure 6 shows array-boundary prioritization for `fn_table`. CAB-FUZZ prioritizes three states according to the associated symbolic variable's constraints: the lowest memory address `fn_table[0]`, the highest memory address `fn_table[36]`, and an arbitrary memory address between them, e.g., `fn_table[13]`.

3.2.2 Loop-boundary Prioritization

Handling a loop can result in state explosion [18]. To avoid it, CAB-FUZZ limits the number of state forks at the same loop to focus on boundary states. Specifically, it focuses on only three states: a state with *no loop execution*, a state with *a single loop execution*, and a state with *the largest number of loop executions*. Figure 6 has a loop whose number of iterations depends on `buf[0]`. Since its values lie between 0 and 246, this loop generates 247 states. To avoid such state explosion, our loop-boundary prioritization method focuses on three kinds of loop executions: 0, 1, and maximum (246) times.

In total, our method generates only 27 states first, `flag_table` (3) × `fn_table` (3) × the loop (3), including the boundary condition causing a crash, `buf[2] == 36`.

4 Implementation

We implemented CAB-FUZZ by extending S2E [10]. In particular, we focused on crashing *Windows device drivers*, which are popular and complex commodity COTS kernel binaries. In total, we wrote around 2,000 lines of new code (mixed with C/C++, Lua, and Python).

4.1 Synthetic Symbolization

We used `NtCreateFile` to obtain the handlers for device drivers. As opposed to using the typical `CreateFile`,

this approach allowed us to access all device drivers, including those of all internal and undocumented devices.

When opening or creating a file object using `NtCreateFile`, we can specify 13 different access rights for the file object [34]. Since we aimed to obtain and test as diverse access rights as possible, we repeatedly invoked `NtCreateFile` in `get_allowed_access` to obtain all possibly allowed permission accesses.

4.2 On-the-fly Symbolization

Target API. To detect device driver bugs with on-the-fly symbolization, we interpose the `NtDeviceIoControlFile` function, which is the lowest user-level internal API for communicating with the kernel devices. Any user-space process attempting to access a device driver eventually calls the function, so hooking it allows us to test all the device drivers used during the on-the-fly symbolization phase. The below half of Figure 3 shows the specification of `NtDeviceIoControlFile`, and CAB-FUZZ symbolizes `in_buf` and `ctrl_code` on-the-fly.

Fulfilling Pre-context. We inferred the pre-context of `NtDeviceIoControlFile` by running real user-space programs using this function during their normal execution. We tried to find such programs with an assumption: system management and antivirus software would use it because they frequently access device drivers. Finally, we found 16 programs (e.g., `dxdiag.exe` and `perfmon.msc`) accessing 15 different drivers (e.g., KsecDD and WMIDataDevice) during their execution[1]. We used these target programs to test the corresponding device drivers during the on-the-fly symbolization phase (§5.2).

4.3 Boundary-state Prioritization

Prioritizing Array Boundaries. For a symbolic memory array, CAB-FUZZ estimates its lower and upper boundary addresses and one arbitrary address between them. CAB-FUZZ uses the `getRange` method of the `klee::Solver` to compute these boundary addresses [6]. This method receives an expression as input and returns a pair of the minimum and maximum values of the expression. Since `getRange` is computationally heavy, instead of invoking this function in every symbolic memory access, CAB-FUZZ proceeds only if the targeted memory has triggered a state forking at one point in the past. Specifically, if state forking has never been triggered, CAB-FUZZ does not perform any prioritization for the memory, as we found that such memory usually has only one concrete value. If the state forking is triggered at the same location, CAB-FUZZ performs prioritization when it observes the memory again later.

Prioritizing Loop Boundaries. CAB-FUZZ focuses on three states of each loop: no, single, and maximum execution (§3.2.2). However, identifying how many times a loop will be executed is difficult because it varies according to input variables and compiler optimization techniques (e.g., loop unrolling [45]). We develop a practical loop-boundary prioritization technique that does not suffer from variable loop conditions. Whenever CAB-FUZZ encounters a loop, it first generates two forking states: no and single iteration of the loop. Then, to get the maximum number of loop executions, it repeatedly forks and kills states until it observes the last state forking, which would be the maximum because CAB-FUZZ concretely and sequentially executes the loop until it terminates. During state forking, CAB-FUZZ does not call the solver to minimize overhead; it calls the solver only when generating test cases. Also, we confirmed that killing unnecessary loop states had negligible performance overhead.

[1] Due to the space limit, we do not enumerate all of them.

4.4 Analyzing Crashes

CAB-FUZZ generated many inputs that crashed the Windows kernel, but a large portion of them may not be unique vulnerabilities that require in-depth analysis. A typical technique of classifying such crashes is to inspect the call stack at the time of the crash, but it is difficult to identify stack information without debug symbols. More seriously, we found that many memory access violations are delegated to the default exception handler, making it even harder to uniquely identify the call stack information of the kernel thread that actually raised the exception.

To solve this problem, CAB-FUZZ records and inspects the blue screen of death (BSOD) information when the Windows kernel executes the `KeBugCheck*` function to gradually bring down the computer [32]. Specifically, CAB-FUZZ uses the function's *BugCheckCode* value representing a BSOD reason and *instruction address* where the exception occurred to differentiate crashes. CAB-FUZZ treats two crashes as different when (1) they have different BugCheckCode values or (2) they have the same BugCheckCode value, but their instruction addresses belong to different functions.

5 Evaluation

We evaluate the effectiveness of CAB-FUZZ in finding security vulnerabilities in the Windows device drivers. Table 1 summarizes all new unique crashes discovered by CAB-FUZZ. In general, our evaluation consists of two categories targeting synthetic symbolization (§5.1) and on-the-fly symbolization (§5.2). In particular, our evaluation aims at answering the following questions:

- Per synthetic symbolization, how efficiently did CAB-FUZZ detect the known vulnerability (Figure 1) compared to the conventional concolic testing tool? (§5.1.1)
- Per synthetic and on-the-fly symbolization, how many new unique crashes did CAB-FUZZ discover? (§5.1.2 and §5.2.1)

	# of Crashes			
	Total	**Synthetic (Prioritization)**		**On-the-fly**
		Off	**On**	
NDIS[†,§]	11	5	10	-
SrvAdmin[†,§]	4	4	4	-
NSI[§]	2	2	2	0
ASYNCMAC[†,§]	1	1	1	-
FileInfo[§]	2	0	0	2
ehdrv[†,§]	1	0	0	1
Total	**21**	**12**	**17**	**3**

[†]: Windows 7, [§]: Windows Server 2008

Table 1: The list of newly discovered unique crashes by CAB-FUZZ among the 274 drivers we tested. The total number of discovered unique crashes is smaller than the summation of the other three columns (two synthetic and one on-the-fly cases) because we removed duplicate crashes and only counted the unique crashes.

- Per synthetic and on-the-fly symbolization, what particular characteristics did newly discovered crashes exhibit? (§5.1.3 and §5.2.2)

Experimental Setup. Our experiments were performed on 3 GHz 8-core Intel Xeon E5 CPU with 48 GB of memory. We ran CAB-FUZZ with the latest versions of Windows 7 and Windows Server 2008 as of April 2016. For example, two of the drivers for which CAB-FUZZ found crashes, NDIS and SrvAdmin, were updated in December 2015 and October 2015, respectively. The detailed configuration setting for CAB-FUZZ is further described in each subsection if required.

5.1 Synthetic Symbolization

To show the effectiveness of the synthetic symbolization and boundary prioritization techniques, we carried out the following two experiments. First, to see if the implementation of CAB-FUZZ can address the challenges (especially in handling state explosion), we applied boundary-state prioritization techniques to the known NDProxy vulnerability and compared the result before applying (§5.1.1). Next, we describe our experiences in applying CAB-FUZZ to discover new crashes in the Windows kernel driver using synthetic symbolization techniques (§5.1.2). Further, we manually analyzed all unique crashes newly discovered by CAB-FUZZ (§5.1.3).

Configuration. We configured CAB-FUZZ to target 186 and 88 kernel device drivers on Windows 7 and Windows Server 2008, respectively (274 drivers in total). Among them, CAB-FUZZ detected *six device drivers* with 21 unique crashes (Table 1). For each device driver, we specified `ctrl_code` and `in_buf` as symbolic variables (shown in Figure 3). It is worth nothing that due to the space limit of this paper, we have only presented the results with a random search strategy, which showed the best performance overall compared to other depth-first and breadth-first search strategies. Since the random search algorithm may produce different evaluation results due to its random nature, we ran it five times per evaluation and computed the average. In addition, when we found the same crash of the same driver in Windows 7 and Windows Server 2008, we further tested it in Windows 7 only since it is the recent version.

Prioritization	**Time (s)**	**#States**
None	7,196	384,817
Loop boundary	516	30,604
Array boundary	2	78
Both	2	78

Table 2: The effectiveness of boundary-state prioritization techniques (based on the synthetic symbolization) to detect the NDProxy vulnerability: **Time** shows the elapsed time and **#States** shows the number of explored states to detect the vulnerability.

5.1.1 Detecting Known Vulnerability

We measured the time taken to find the NDProxy vulnerability (Figure 1) before and after applying the prioritization techniques. We also measured the number of program states that need to be explored to find the vulnerability.

When both array- and loop-boundary prioritization techniques were applied, CAB-FUZZ found the NDProxy vulnerability within 2 seconds (Table 2). It took 2 seconds with the array-boundary prioritization and 516 seconds with the loop-boundary prioritization if each technique was individually applied. The array-boundary prioritization is more effective than the loop-boundary prioritization in the case of the NDProxy vulnerability because the state related to the crash (i.e., `buf[2] == 36`) is quickly created by the array-boundary prioritization technique, as shown in Figure 1.

However, when none of prioritization techniques were applied, it took 7,196 seconds to find the vulnerability. This significant slowdown is caused by the huge number of states that need to be covered in order to find the vulnerability—384,817 states in total, which is 4,934 times larger than the number of states when both were applied.

5.1.2 Newly Discovered Crashes

To determine the effectiveness of our synthetic symbolization with and without prioritization techniques, we applied CAB-FUZZ to all kernel device drivers in Windows 7 and Windows Server 2008. In total, CAB-FUZZ found 18 new unique crashes from four different device drivers, as shown in Table 1. Specifically, the prioritization techniques allowed CAB-FUZZ to detect six more unique crashes while missing one unique crash. Thus, we believe this technique is effective in practice.

Driver		No prioritization			Prioritization		
	#Crash	Time (s)	#States	Mem. (MB)	Time (s)	#States	Mem. (MB)
NDIS	1	837	151	5,537	287	58	4,813
	2	871	156	5,545	467	86	4,971
	3	1,763	271	7,690	617	124	5,027
	4	5,066	637	14,946	824	171	5,461
	5	8,682	1,180	22,768	1,202	214	6,093
	6	-	-	-	1,930	306	7,980
	7	-	-	-	4,381	586	9,781
	8	-	-	-	4,977	637	10,376
	9	-	-	-	5,018	642	10,377
	10	-	-	-	6,056	704	10,893
SrvAdmin	1	1	23	4,321	2	23	4,325
	2	3	54	4,359	6	71	4,401
	3	51	126	4,464	51	126	4,476
	4	1,892	2,319	15,321	657	953	5,390
NSI	1	1	2	4,356	1	2	4,357
	2	1,951	7,622	5,979	1,092	1,952	5,843

Table 3: Detailed experiment results of the four kernel device drivers tested by CAB-FUZZ with and without prioritization techniques: **#Crash** represents how many crashed observed during experiments; **Time** represents the elapsed time; **#States** represents the number of explored states; and **Memory** represents the consumed memory to detect each crash. All values are averaged over five runs.

5.1.3 Effectiveness of Boundary-state Prioritization

To clearly understand the effectiveness of our prioritization techniques, we manually analyzed why CAB-FUZZ without our prioritization techniques cannot detect the six unique crashes and what is the root cause of its slowdown. Note that our prioritization techniques were ineffective to ASYNCMAC (elapsed time and memory consumption were almost the same,) so we skipped analyzing it in depth. Also, we were not able to test their effectiveness with other device drivers because CAB-FUZZ was not able to detect their crashes. Table 3 represents how many crashes were observed during our evaluation along with elapsed time, the number of tested states, and consumed memory. All results are averaged over five runs. Note that, because we use a random search strategy, it is difficult to directly compare each crash.

NDIS. The six crashes that the prioritization technique detected were due to input buffers whose values were used as offsets of a symbolic array. When there were no routines to check the range of input buffer values or the values were incorrect, crashes were generated due to invalid offsets. However, without prioritization, CAB-FUZZ was unable to reproduce it due to memory exhaustion.

Among the five crashes that CAB-FUZZ with prioritization was able to generate but CAB-FUZZ without prioritization was unable to do, we explain a crash at `ndisNsiGetNetworkInfo` function of ndis.sys in detail. The function had a symbolic memory array using `in_buf[5]` as an offset, but did not have any routine to check its value. As a result, when the symbolic array pointed to invalid memory and there was a write attempt to the memory, a crash occurred. This happened when the value of `in_buf[5]` was at the boundary condition: whether it was larger than or equal to `0xbc0`, but, without prioritization, CAB-FUZZ could not generate this state due to a lack of available memory (it concretized ∼30 values of `in_buf[5]` before termination.)

On the other hand, the single crash that CAB-FUZZ with prioritization could not detect was due to the loop-boundary prioritization technique. We found that the `ndisNsiGetInterfaceRodEnumObject` function of ndis.sys generated a crash when it ran a loop four times with a specific condition. Note that our loop-boundary prioritization technique runs a loop 0, 1, or a maximum number of times, so it cannot cover such a specific case. To confirm it, we applied CAB-FUZZ only with the array-boundary prioritization to NDIS. We could trigger the specific case also, though it took about one hour longer.

SrvAdmin. We analyzed SrvAdmin and confirmed that the 2.9× slowdown of CAB-FUZZ without prioritization was due to the state explosion caused by a specific loop located at the `SvcAliasEnumApiHandler` function of srvnet.sys. This loop was not related to the crash we found, but it generated 8,285 states that were approximately 20% of the entire states (41,279) of SrvAdmin. With the loop-boundary prioritization, CAB-FUZZ could postpone less important states, so it detected the crash earlier.

NSI. We analyzed NSI and confirmed that our prioritization techniques made CAB-FUZZ detect the two unique crashes 1.8× faster. While symbolic arrays or loops were not directly related to these crashes, we found that prioritization techniques helped concolic testing avoid the state explosion, so that it kept exploring the program states and finally reached the vulnerable program state.

5.2 On-the-Fly Symbolization

We evaluate the effectiveness of on-the-fly symbolization. We summarize the new crashes the on-the-fly technique detected (§5.2.1) and analyze them in detail to show how this technique was able to detect them (§5.2.2).

5.2.1 Newly Discovered Crashes

Overall, CAB-FUZZ identified three unique crashes using on-the-fly symbolization (Table 1). Note that the crashes found by the two techniques were not overlapped because (1) the on-the-fly technique was unable to test some drivers (NDIS, SrvAdmin, and ASYNCMAC) because we had no reference applications accessing them and (2) some crashes (in NSI) were triggered only if they had improper pre-contexts. Therefore, we believe both techniques are complementary to each other.

5.2.2 Effectiveness of On-the-fly Symbolization

To figure out how the on-the-fly technique helps find a vulnerability, we manually analyzed three crashes that CAB-FUZZ found in FileInfo and ehdrv device drivers.

	# of Crashes		
	Total	**Synthetic**	**On-the-fly**
WMIDataDevice	2	1	1
TCP	3	3	0
Total	**5**	**4**	**1**

Table 4: The crashes of Windows XP CAB-FUZZ found.

FileInfo. We found two reasons why the on-the-fly technique was able to find these cases and why synthetic symbolization was not. First, FileInfo was loaded only when a certain application started (e.g., `perfmon.msc`). Second, FileInfo sanitized an input buffer size at an early stage; it should be 12. Running `perfmon.msc` satisfied both conditions for the on-the-fly technique, but a synthetic program was unable to do that.

ehdrv. ehdrv was a third-party driver installed by ESET Smart Security 9, which was used by `SysInspector.exe` of the vendor. The on-the-fly technique detected a memory corruption crash of ehdrv on Windows 7 by running `SysInspector.exe` before symbolization. In contrast, the synthetic technique cannot detect it because ehdrv had a security feature: it was only accessible by an authorized process like `SysInspector.exe`, which cannot be satisfied by a synthetic program.

5.3 Fourteen-Year-Old Bugs

We applied CAB-FUZZ to the latest version of Windows XP (April 2014) and found five unique crashes (Table 4). Among them, a crash of WMIDataDevice and all three crashes of TCP were also observed in the initial version of Windows XP (August 2001), implying *nobody detected them for about 14 years*.

6 Discussion

In this section we explain some limitations of CAB-FUZZ.

Boundary-state Prioritization. Our boundary-state prioritization methods assume that the symbolic memory under consideration stores data such that values between boundaries are less important; that is, we sacrifice some completeness for efficient detection. However, if the symbolic memory is related to control flow (e.g., jump table and virtual function table), we should consider all the values to maintain code coverage. To solve this problem, we plan to adopt static analysis in our system. Whenever it detects a symbolic memory array, it performs static analysis to know whether the symbolic array stores instruction addresses for indirect calls or jumps. In such a case, it checks all the values of the symbolic array to enhance code coverage. Also, our methods cannot handle data structures with undefined size. We plan to enhance CAB-FUZZ to support this in the future. For example, we can adopt UC-KLEE [14, 39, 40]-like approaches.

On-the-fly Symbolization. Our on-the-fly approach is a best-effort approach. If we cannot find programs constructing pre-contexts for vulnerable functions, it cannot crash them. Thus, this approach is not suitable for detecting the security vulnerabilities of rarely used functions. To detect vulnerabilities in such functions, one would need to run synthetic and on-the-fly testing in parallel.

Manual efforts. Currently, we manually specify a target API, `NtDeviceIoControlFile`, for the both synthetic and on-the-fly symbolizations, and programs constructing pre-contexts for the on-the-fly symbolization. In the future, we will explore how to automate both phases for enhancing CAB-FUZZ's scalability.

7 Related Work

In this section, we introduce previous work related to CAB-FUZZ. Among a large number of studies on symbolic and concolic execution, we focus on four research topics closely related to CAB-FUZZ: (1) binary-level symbolic execution, (2) kernel and device driver testing, (3) boundary value analysis, (4) overflow detection, and (5) lazy initialization.

Binary-level Symbolic Execution. Symbolic execution was originally designed to work with source code [4, 7, 12, 16, 29, 30], and extended to test binary programs lacking source code and detailed debug information (e.g., proprietary software and malware). SAGE [3, 17] is the earliest effort to apply symbolic execution to binary programs and many schemes such as SmartFuzz [36], LESE [42], IntScope [47], S2E [10], FuzzBALL [2, 31], Mayhem [9], MegaPoint [1], and DIODE [44] follow it. Among them, only S2E and FuzzBALL are designed to test OS kernels, while FuzzBALL does not support Windows binaries. Consequently, S2E is the only scheme that we can directly compare with CAB-FUZZ.

Kernel and Device Driver Testing. CAB-FUZZ is designed to test COTS OSes and device drivers. To the best of our knowledge, only a few studies apply concolic execution to OSes and device drivers. Yang et al. [50] use their EXE system [7] to create a symbolic disk for Linux file system testing. Their system relies on file system code instrumentation to create the symbolic disk, so it cannot be applied to COTS OSes directly.

DDT [27] is a QEMU-based system to test closed-source binary device drivers for Windows, which became a part of S2E [10]. It can test device drivers without real hardware by creating symbolic hardware (e.g., network interface card and sound card). However, without manual annotations and configurations, it neither identifies device driver interfaces due to lack of kernel symbols nor meets conditions to initialize them.

SymDrive [41] is an S2E-based system to test Linux and FreeBSD drivers without devices, while overcoming the limitation of DDT. It uses a static analysis to auto-

matically identify driver code's key features such as entry point and loop, so, unlike DDT, it can correctly initialize device drivers without requiring manual effort. However, it also relies on source code instrumentation, so it cannot be applied to COTS OSes lacking debug information.

Trinity [24] and IOCTL Fuzzer [15] are system call fuzzers based on Linux and Windows, respectively. Before fuzzing a certain system call, they also try to construct pre-contexts, which is similar to CAB-FUZZ's on-the-fly technique. The key difference here is that CAB-FUZZ symbolizes the input, but these previous efforts randomly mutate input values only once. Thus, they have difficulties in detecting sophisticated conditions to trigger vulnerabilities.

Unlike the other systems described here, CAB-FUZZ does not rely on source code analysis or instrumentation, so it can be freely applied to COTS OSes. Furthermore, it does not suffer from the initialization problem thanks to its on-the-fly concolic testing.

Boundary Value Analysis. Several researchers have proposed boundary value analysis techniques [22, 23, 26, 38] to maximize branch coverage. For example, ADSE [22, 23] checks constraints at every path and loop and augments conditions to figure out which conditions generate maximum test cases. These approaches can detect the correct boundary conditions; however, the overall conditions will easily explode if we apply them to complex software, e.g., OSes. In contrast, CAB-FUZZ creates only two boundary states plus one arbitrary state for each symbolic array and loop such that it practically mitigates the state explosion problem.

Overflow Detection. CAB-FUZZ focuses on the boundaries of symbolic memories and loops because such boundaries could trigger stack or heap over/underflows. Several studies attempt to specialize symbolic execution to detect overflow and underflows. IntScope [47] and SmartFuzz [36] use symbolic execution to detect integer overflows. In addition, SmartFuzz covers integer underflows, narrowing conversions, and signed/unsigned conversions. Dowser [19] considers a buffer in a loop to detect its overflows and underflows. DIODE [44]'s goal is to find integer overflow errors at target memory locations. It uses a fine-grained dynamic taint analysis to identify all memory allocation sites, extracts target and branch constraints from instrumented execution, solves the constrains, and performs goal-directed conditional branch enforcement.

Although these methods work well, they rely on heavy static analysis and/or taint analysis to detect specific integers or buffers that could result in overflows. In contrast, CAB-FUZZ does not use such complicated analysis techniques when detecting boundaries, so it is more lightweight and practical than the previous techniques.

Lazy Initialization. CAB-FUZZ's on-the-fly concolic testing is a kind of lazy initialization technique [25, 49] that defers the initialization of memory or a data structure until it is actually used. Firmalice [43] is a binary analysis framework to analyze the firmware of embedded devices. It uses a lazy initialization technique to test memory because it does not know which code needs to be executed to initialize specific memory regions. When Firmalice detects a memory read from uninitialized memory during analysis, it pauses the execution and conducts the following procedures. First, it identifies other procedures that contain direct writes to the memory. Next, it labels the procedures as initialization procedures. Last, it duplicates the state: (1) resumes the execution without any modification to avoid possible crashes and (2) runs the initialization procedures before resuming the execution. However, a static program analysis is necessary to detect such initialization procedures.

UC-KLEE [14, 39, 40] directly tests individual functions instead of the whole program to improve scalability. To cope with missing pre-contexts of individual functions, it automatically generates symbolic inputs using lazy initialization. However, it still suffers from false positives due to invariants of data structures, state machines, and APIs, so it relies on manual annotations to reduce them.

On the contrary, CAB-FUZZ's on-the-fly concolic testing neither requires sophisticated static program analysis nor suffers from false positives. Also, it can be fully automated because it uses the real execution procedures of a target program.

8 Conclusion

In this paper, we presented a practical concolic testing tool, CAB-FUZZ, to analyze COTS OSes. CAB-FUZZ introduced two new memory symbolization techniques—synthetic symbolization and on-the-fly symbolization—allowing us to analyze COTS OSes without debug information and pre-contexts. It employed two boundary-state prioritization techniques: array- and loop-boundary prioritization, allowing us to prioritize potentially vulnerable paths. Evaluation results showed that CAB-FUZZ can detect 21 undisclosed unique crashes on Windows 7 and Windows Server 2008 while avoiding the state explosion problem.

Acknowledgements. We thank the anonymous reviewers and our shepherd, Mihai Christodorescu, for their helpful feedback. This research was supported by the NSF award DGE-1500084, CNS-1563848, CRI-1629851 ONR under grant N000141512162, DARPA TC program under contract No. DARPA FA8650-15-C-7556, DARPA XD3 program under contract No. DARPA HR0011-16-C-0059, and ETRI MSIP/IITP[B0101-15-0644].

References

[1] AVGERINOS, T., REBERT, A., CHA, S. K., AND BRUMLEY, D. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)* (Hyderabad, India, May–June 2014).

[2] BABIĆ, D., MARTIGNONI, L., MCCAMANT, S., AND SONG, D. Statically-Directed Dynamic Automated Test Generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)* (Toronto, Canada, July 2011).

[3] BOUNIMOVA, E., GODEFROID, P., AND MOLNAR, D. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)* (2013), pp. 122–131.

[4] BOYAPATI, C., KHURSHID, S., AND MARINOV, D. Korat: Automated Testing Based on Java Predicates. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)* (2002).

[5] BUGRARA, S., AND ENGLER, D. Redundant State Detection for Dynamic Symbolic Execution. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)* (San Jose, CA, June 2013).

[6] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (San Diego, CA, Dec. 2008).

[7] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., AND ENGLER, D. R. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)* (Alexandria, VA, Oct.–Nov. 2006).

[8] CADAR, C., AND SEN, K. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the ACM 56*, 2 (2013), 82–90.

[9] CHA, S. K., AVGERINOS, T., REBERT, A., AND BRUMLEY, D. Unleashing Mayhem on Binary Code. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)* (San Francisco, CA, May 2012).

[10] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2E: A Platform for In Vivo Multi-Path Analysis of Software Systems. In *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Newport Beach, CA, Mar. 2011).

[11] CVE. CVE-2013-5065, 2013. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-5065.

[12] DENG, X., LEE, J., AND ROBBY. Bogor/Kiasan: A k-bounded Symbolic Execution for Checking Strong Heap Properties of Open Systems. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2006).

[13] DOLAN-GAVITT, B., HULIN, P., KIRDA, E., LEEK, T., MAMBRETTI, A., ROBERTSON, W., ULRICH, F., AND WHELAN, R. LAVA: Large-scale Automated Vulnerability Addition. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)* (San Jose, CA, May 2016).

[14] ENGLER, D., AND DUNBAR, D. Under-Constrained Execution: Making Automatic Code Destruction Easy and Scalable. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)* (London, UK, July 2007).

[15] ESAGE LAB. IOCTL Fuzzer: Windows Kernel Driver Fuzzer. https://code.google.com/archive/p/ioctlfuzzer/.

[16] GODEFROID, P. Compositional Dynamic Test Generation. In *Proceedings of the 34th ACM Symposium on Principles of Programming Languages (POPL)* (Nice, France, Jan. 2007).

[17] GODEFROID, P., LEVEN, M. Y., AND MOLNAR, D. Automated Whitebox Fuzz Testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2008).

[18] GODEFROID, P., AND LUCHAUP, D. Automatic Partial Loop Summarization in Dynamic Test Generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)* (Toronto, Canada, July 2011).

[19] HALLER, I., SLOWINSKA, A., NEUGSCHWANDTNER, M., AND BOS, H. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Proceedings of the 22th USENIX Security Symposium (Security)* (Washington, DC, Aug. 2013).

[20] INTEL. Introduction to Intel Memory Protection Extensions. https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions.

[21] "J00RU" JURCZYK, M. Windows X86-64 System Call Table (NT/2000/XP/2003/Vista/2008/7/2012/8). http://j00ru.vexillium.org/ntapi_64/.

[22] JAMROZIK, K., FRASER, G., TILLMAN, N., AND DE HALLEUX, J. Generating Test Suites with Augmented Dynamic Symbolic Execution. In *Proceedings of International Conference on Tests and Proofs* (2013).

[23] JAMROZIK, K., FRASER, G., TILLMANN, N., AND HALLEUX, J. D. Augmented Dynamic Symbolic Execution. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2012).

[24] JONES, D. Trinity: A Linux System Call Fuzz Tester. http://codemonkey.org.uk/projects/trinity/.

[25] KHURSHID, S., PĂSĂREANU, C. S., AND VISSER, W. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2003).

[26] KOSMATOV, N., LEGEARD, B., PEUREUX, F., AND UTTING, M. Boundary Coverage Criteria for Test Generation from Formal Models. In *Proceedings of 15th International Symposium on Software Reliability Engineering (ISSRE)* (2004).

[27] KUZNETSOV, V., CHIPOUNOV, V., AND CANDEA, G. Testing Closed-Source Binary Device Drivers with DDT. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC)* (Boston, MA, June 2010).

[28] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, Colorado, Oct. 2014).

[29] MAJUMDAR, R., AND SEN, K. Hybrid Concolic Testing. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)* (Minneapolis, MN, May 2007).

[30] MARINESCU, P. D., AND CADAR, C. make test-zesti: A Symbolic Execution Solution for Improving Regression Testing. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)* (Zurich, Switzerland, June 2012).

[31] MARTIGNONI, L., MCCAMANT, S., POOSANKAM, P., SONG, D., AND MANIATIS, P. Path-Exploration Lifting: Hi-Fi Tests for Lo-Fi Emulators. In *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (London, UK, Mar. 2012).

[32] MICROSOFT. KeBugCheckEX routine (Windows Drivers). https://msdn.microsoft.com/en-us/library/windows/hardware/ff551961(v=vs.85).aspx.

[33] MICROSOFT. Methods for Accessing Data Buffers. https://msdn.microsoft.com/en-us/library/windows/hardware/ff554436(v=vs.85).aspx.

[34] MICROSOFT. NtCreateFile Function (Windows). https://msdn.microsoft.com/en-us/library/bb432380(v=vs.85).aspx.

[35] MICROSOFT. NtDeviceIoControlFile function (Windows). https://msdn.microsoft.com/en-us/library/ms648411(v=vs.85).aspx.

[36] MOLNAR, D., LI, X. C., AND WAGNER, D. A. Dynamic Test Generation To Find Integer Bugs in x86 Binary Linux Programs. In *Proceedings of the 18th USENIX Security Symposium (Security)* (Montreal, Canada, Aug. 2009).

[37] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M. K., AND ZDANCEWIC, S. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Dublin, Ireland, June 2009).

[38] PANDITA, R., XIE, T., TILLMANN, N., AND DE HALLEUX, J. Guided Test Generation for Coverage Criteria. In *Proceedings of IEEE International Conference on Software Maintenance (ICSM)* (2010).

[39] RAMOS, D., AND ENGLER, D. Practical, Low-effort Verification of Real Code using Under-constrained Execution. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)* (Snowbird, UT, July 2011).

[40] RAMOS, D. A., AND ENGLER, D. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *Proceedings of the 24th USENIX Security Symposium (Security)* (Washington, DC, Aug. 2015).

[41] RENZELMANN, M. J., KADAV, A., AND SWIFT, M. M. SymDrive: Testing Drivers without Devices. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Hollywood, CA, Oct. 2012).

[42] SAXENA, P., POOSANKAM, P., MCCAMANT, S., AND SONG, D. Loop-Extended Symbolic Execution on Binary Programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)* (Chicago, IL, July 2009).

[43] SHOSHITAISHVILI, Y., WANG, R., HAUSER, C., KRUEGEL, C., AND VIGNA, G. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2015).

[44] SIDIROGLOU-DOUSKOS, S., LAHTINEN, E., RITTENHOUSE, N., PISELLI, P., LONG, F., KIM, D., AND RINARD, M. Targeted Automatic Integer Overflow Discovery Using Goal-Directed Conditional Branch Enforcement. In *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Istanbul, Turkey, Mar. 2015).

[45] SLOWINSKA, A., STANCESCU, T., AND BOS, H. Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2011).

[46] STEPHENS, N., GROSEN, J., SALLS, C., DUTCHER, A., WANG, R., CORBETTA, J., SHOSHITAISHVILI, Y., KRUEGEL, C., AND VIGNA, G. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2016).

[47] WANG, T., WEI, T., LIN, Z., AND ZOU, W. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2009).

[48] WANG, X., ZHANG, L., AND TANOFSKY, P. Experience Report: How is Dynamic Symbolic Execution Different from Manual Testing? A Study on KLEE. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)* (Baltimore, MD, July 2015).

[49] XIE, Y., AND AIKEN, A. Scalable Error Detection using Boolean Satisfiability. In *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages (POPL)* (Long Beach, CA, Jan. 2005).

[50] YANG, J., SAR, C., TWOHEY, P., CADAR, C., AND ENGLER, D. Automatically Generating Malicious Disks using Symbolic Execution. In *Proceedings of the 27th IEEE Symposium on Security and Privacy (Oakland)* (Oakland, CA, May 2006).

Log-Structured Non-Volatile Main Memory

Qingda Hu* Jinglei Ren Anirudh Badam Jiwu Shu* Thomas Moscibroda
*Tsinghua University Microsoft Research

Abstract

Emerging non-volatile main memory (NVMM) unlocks the performance potential of applications by storing persistent data in the *main memory*. Such applications require a *lightweight* persistent transactional memory (PTM) system, instead of a *heavyweight* filesystem or database, to have fast access to data. In a PTM system, the memory usage, both capacity and bandwidth, plays a key role in dictating performance and efficiency. Existing memory management mechanisms for PTMs generate high memory fragmentation, high write traffic and a large number of persist barriers, since data is first written to a log and then to the main data store.

In this paper, we present a *log-structured* NVMM system that not only maintains NVMM in a compact manner but also reduces the write traffic and the number of persist barriers needed for executing transactions. All data allocations and modifications are appended to the log which becomes the location of the data. Further, we address a unique challenge of log-structured memory management by designing a *tree*-based address translation mechanism where *access granularities* are flexible and different from *allocation granularities*. Our results show that the new system enjoys up to 89.9% higher transaction throughput and up to 82.8% lower write traffic than a traditional PTM system.

1 Introduction

Emerging byte-addressable non-volatile main memory (NVMM), e.g., 3D XPoint [23], PCM [43, 27], STT-RAM [3, 25] and ReRAM [2], enables persistent data to be stored in *main memory*. This leads to an architecture where applications directly access persistent data via CPU load/store instructions [50, 10, 49, 37, 44, 18, 41, 55]. Such an architecture lowers latency not only due to the significantly higher performance of NVMM compared to SSDs, but also due to the fact that the system software is removed from the critical path of persistent-data accesses [36, 13, 9, 55]. Applications that use NVMM typically employ a lightweight persistent transactional memory (PTM) system [50, 10, 22, 56, 34, 9, 18, 24], instead of a traditional file system or database, to have fast access to NVMM data.

Memory usage, both capacity and bandwidth, is crucial for the performance and efficiency of PTM systems. DRAM-style memory management used by existing PTM systems to manage NVMM leads to a high amount of fragmentation that can cause wastage of over 50% space [46]. Moreover, existing transactional mechanisms used by PTM systems lead to excessive write traffic as they require all new data to be written twice – once to the log, and once to the main data region, referred to as *home space* of data. The redundant writes not only increase memory bandwidth usage but also wear out the NVMM device faster. Further, these writes need to be persisted using expensive barriers in a synchronous manner which increase the latency of transactions.

In this paper, we present a new *log-structured* memory management model for NVMM systems. This model eliminates *dichotomy* of NVMM data in the home space and a separate log area. We unify the home space and the log area by organizing the whole NVMM solely in the form of logs, which also act as the home space. Our design effectively reduces fragmentation, incorporates wear-leveling, and optimizes for the write traffic and persist barriers. Fragmentation is minimal because memory allocation becomes an immediate append to the end of a log, and freed up areas can be moved and consolidated [45, 46] to further reduce fragmentation. Besides, NVMM bandwidth consumption, write wear and the number of persist barriers are reduced because there is no need to write data *separately* to both the traditional home space and the log.

Applications using our system view NVMM in the same way as the traditional systems, but a runtime address mapping mechanism is employed to translate application addresses to log offsets. We refer to the applications' view of NVMM as the *virtual* home space. Such address mappings are fully cached in DRAM, and can be consistently restored from the log after a crash.

Another *key contribution* of this work is the design and implementation of a practical tree data structure for the home to log address mapping in our system. While log-structured approaches have been explored in different domains, such as filesystems [45, 52], databases [48, 46, 4] and object stores [31, 46], log-structured NVMM faces a *unique challenge* of address mapping overhead. Unlike existing log-structured systems, we need to present a flat address space where allocation granularities are *not* the same as access granularities.

A data structure that can support creation of mappings at access time as opposed to allocation time is required.

*Work done while Q. Hu was an intern at Microsoft Research. Affiliated with Dept. of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology (TNLIST).

This is because memory stores can target *arbitrary* addresses and lengths that may not be indicated at allocation time. We show in this paper that a *tree structure* is well suited for such a requirement. Meanwhile, NVMM is orders of magnitude faster than SSDs, so that address mapping performance could become a bottleneck if not designed well. For SSDs, data access latencies dwarf address translation overhead, but that is not the case with NVMM. Hence, we revisit the address mapping issue of log-structured designs for NVMM systems.

A naive tree data structure requires $O(\log n)$ operations per memory access which can be prohibitive when n is large. Moreover, trees require expensive balancing operations to achieve such time complexity. We design key optimizations to a tree structure for log-structure NVMM to reduce address translation overhead: **(1)** *Two-layer mapping*. The whole home space is first divided into static fixed-length *partitions* so that data can be routed to such a partition (or more partitions) in $O(1)$ time. In this way, the average number of nodes in a partition-local tree is much smaller than a huge tree covering the whole address space. **(2)** *Skip-list trees*. We use the skip list [42] for second-layer trees. The main benefit is that they probabilistically balance at insert time to avoid rebalancing operations, which are costly and largely impair parallelism. **(3)** *Group update*. If consecutive writes target contiguous addresses, we merge them and update the tree only once. **(4)** *Tree node cache*. We observe that memory accesses have locality so that caching recently visited tree nodes can avoid many full tree lookup paths starting from the root node.

We also present mechanisms to control the overhead of log cleaning needed for compaction, and speed up the recovery process. NVMM logs can be processed in parallel on recovery, which helps rebuild address mappings for 10 GB NVMM in 3.0 seconds.

Overall, we make the following contributions:

- A new log-structured design to eliminate the dichotomy between the data and the transactional log for PTMs. We identify the crucial difference between existing log-structured systems and the kind needed for NVMMs where access granularities are not identical to allocation granularities.
- A novel tree-based address mapping mechanism that meets the above requirement. To the best of our knowledge, we are the first to demonstrate the practicality of employing such a well optimized tree structure in a log-structured NVMM system.
- An implementation of the above ideas by modifying TinySTM [16]. Under various workloads, log-structured NVMM achieves 55.3% more throughput and 72.2% less write wear than a traditional PTM on average, when the usage of NVMM is over 90% and the log cleaning overhead takes place.

2 Background and Motivation

Current PTMs typically derive their memory management design from that for DRAM. Data is referenced using load and store instructions on native virtual memory offsets, and memory allocations are managed by an allocator such as Hoard [6] adopted by Mnemosyne [50], and jemalloc [15] adopted by Intel's NVML [22] and Oracle's NVM Direct [39]. However, the following problems arise in such systems.

Fragmentation of NVMM space. There are two sources of fragmentation in a traditional memory allocator [17, 15]. First is *internal fragmentation*. Take Intel's NVML [22] for example. It aligns any NVMM allocation size to 64 B. If 65 B of NVMM is requested, NVML shall effectively allocate 128 B, including 63 B internal fragmentation. Second is *external fragmentation*. Suppose a 64-B block is freed but has surrounding blocks in use, then it cannot serve any request beyond 64 B. External fragmentation is severe if allocation sizes vary [38]. Experiments [46] have demonstrated that fragmentation can take over 50% of all memory under management. This issue is more critical for NVMM because it holds data for a long term even across reboots.

Garbage collection, in a managed language runtime such as Java or C#, is capable of changing allocated addresses. It can reduce fragmentation but involves object reference analysis and process pauses [20]. Since NVMM is slower and larger than DRAM, the cost of object reference analysis and pauses will be prohibitive.

In contrast, a log-structured approach easily avoids internal fragmentation because new allocation is compactly appended to the log end. It absorbs external fragmentation by moving allocated data and consolidating free spaces without the need to pause the process.

Excessive NVMM write traffic and barriers. NVMM has limitations in bandwidth and endurance [28] ($10^4 - 10^9$ P/E cycles compared to DRAM's 10^{15} cycles). However, to maintain crash consistency, all NVMM writes must first be logged by PTM at a separate location. Such logging entails redundant NVMM write traffic and extra wear, compared to naive writing.

Figure 1 shows how a log-structured approach can reduce the write traffic and also the number of flushes for a representative transaction. By the pseudo function `map_address`, all addresses within the area are mapped to a new location in the log. Such mapping only involves DRAM writes which are fast and incur no wear on NVMM. This approach saves extra NVMM writes and costly CPU flushes/persist barriers.

Furthermore, the traditional PTM systems use the NVMM bus less efficiently than the log-structured approach, because updates to the home space tend to be sparse and hence have poor cacheline coverage . This

Pseudo source code:

```
void tx_update_title(employee emp, title new_title) {
  tx_begin {
    emp.title = new_title;
  } tx_end;
}
```

Transaction system behaviors: *logging PTM* vs. *log-structured NVMM*

```
in_log_title = append_to_undo_log(emp.title);
flush(in_log_title);
emp.title = new_title; // extra NVMM write
flush(emp.title); // one more flush
```

```
in_log_title = append_to_nvmm_log(new_title);
flush(in_log_title);
map_address(emp->title, in_log_title);
```

Figure 1: In a traditional PTM, objects have to be first logged and the log has to be persisted in NVM using a CPU flush before the transaction can edit the objects. Another CPU flush is needed after the edits complete. In log-structured NVMM, one flush is enough. Since the log entry becomes the new location of data, the extra write is eliminated.

leads to more bus bandwidth consumption when compared to sequentially appending them to the log.

A unique challenge in log-structured NVMM. The challenge of *tree*-based address mapping is a unique one for log-structured NVMM. It has not been seen in existing log-structured systems. Those systems manage data in a form of well-defined elements such as blocks in a filesystem [45, 52], tuples in a database [48, 4] or objects in a key-value store [31, 46], where allocation granularities are the same as access granularities. Such well defined access granularities facilitate a high performance design. For instance, an in-memory hash table can be employed to map elements to their locations in the log, which offers $O(1)$ lookup. In addition, a bloom filter can be applied to improve mapping/index performance in the case that a slow search path exists (e.g., log-structured merge trees [48]).

Unfortunately, such a convenience is missing for NVMM systems. There is no concept of data elements or IDs in bare memory. It is hard to define one in systems that employ a flat address space where accesses can be targeted at any offset with any length. Restrcting block/object-granular accesses lacks flexibility and incurs high costs [51, 16]. Simply setting a fixed and small block size (e.g., tens of bytes) is not viable either, because the metadata to maintain such blocks can be prohibitively large [30, 19]. Furthermore, NVMM is orders of magnitude faster than SSDs, so the address mapping overhead, though traditionally negligible, now stands out. Therefore, we design a more flexible but highly performant scheme, which fragments the address space on demand based on the executed store instructions rather than defining the granularity statically or at data allocation time.

3 Design

This section describes the design of log-structured NVMM (LSNVMM), a user-space library for accessing and managing NVMM.

3.1 Overview

The high-level architecture of LSNVMM is shown in Figure 2. From bottom up, LSNVMM uses DAX [32] through a filesystem that allows *direct* access to physical NVMM device via a memory map. In LSNVMM, the NVMM region is organized into logs (§3.3), and an address mapping mechanism translates virtual home-space addresses to log positions (§3.2). Applications access the NVMM region via our library that interposes all the memory accesses to the region using the address mapping mechanism.

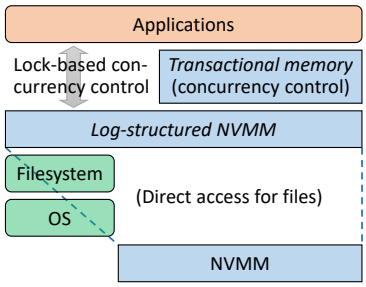


Figure 2: The architecture and system stack of log-structured NVMM.

Interface. Our library offers two main functionalities. One is memory management, with semantics similar to that of C library: `pmalloc` and `pfree` for NVMM allocation and deallocation, respectively[1]. The other functionality of our library is the transaction abstraction that provides crash-consistent data persistence. *All* NVMM data operations are performed via this abstraction, referred to as an NVMM transaction. Within an NVMM transaction, memory loads and stores are instrumented at compile time and treated differently: all stores of the transaction are persisted atomically to the log on NVMM; every load address has to be translated to a

[1]Note that our current design assumes that the persistent region is fixed to a static base address [50, 35]. Doing so enables use of native pointers that remain valid across crashes and reboots. However, special pointer types [10] can be supported easily.

proper position in the log to access the data. Concurrency control of data operations is left to an upper-layer transactional memory (TM) system. It is also possible to use explicit locks for such concurrency control.

Recovery. To achieve efficient address translation, address mappings are stored in DRAM. On a normal process shutdown, we compact the in-DRAM address mappings and other necessary metadata, and flush them to NVMM, so that they can be quickly restored when the process restarts. However, if a system crash happens, the DRAM data is lost. Therefore, we have to rebuild the in-DRAM data structures. To speed up this process, the recovery is performed using thread-level parallelism (more details in §4.5).

3.2 Address Mapping

Using our address mapping mechanism, applications interact with NVMM in much the same way as DRAM to build data structures. They need not change their memory access model that uses flexible regular virtual memory addresses and pointers. However, they have to adopt the transaction interface to make atomic changes to the data structures similar to existing PTM systems. We refer to addresses in applications' view as *home addresses*, and log positions that are hidden from applications as *log addresses*.

We use a tree structure to maintain mappings from home addresses to log addresses. Logically, one node in the tree holds a pair {home address, length} denoting an area in the home space, and the log address that the area is mapped to. The rationale for using a tree instead of a hash table is that, in flat address space based systems, allocation granularities are not identical to access granularities. For instance, an application may allocate a large structure using `pmalloc` but only read/write a small portion of that within transactions. Therefore, we need address translation support for arbitrary accesses that are not aligned with allocated objects.

The efficiency of address mapping is crucial for our system. The latency of traditional log-structured systems is dominated by the disk/SSD latency of data accesses. Also, the granularity of such data accesses is large (e.g., the block size of 512 B) and the frequency is low. However, in our case, NVMM is much faster and more frequently accessed in granularities as small as a few bytes. Hence, it warrants careful design of the address mapping. The time complexity of an operation on the tree is $O(\log n)$. We use several optimizations to reduce the practical cost of such an operation. Figure 3 depicts main data structures to support these optimizations as described below.

Two layers of mapping. The average cost of a tree operation is proportional to the tree height, so our first optimization targets at largely reducing the tree height. This

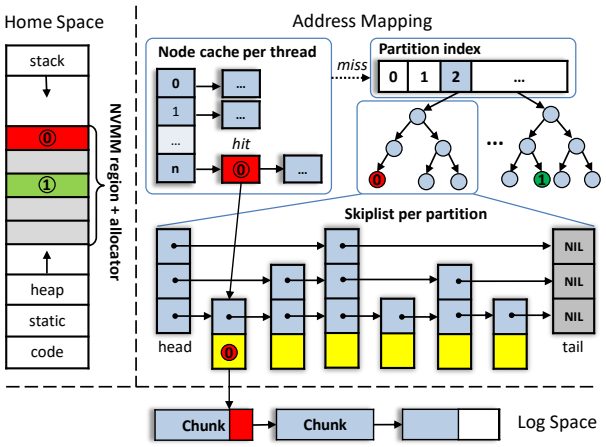


Figure 3: The spaces and address mappings in LSNVMM. Access to ⓪ is a hit in the tree node cache, and access to ① is a miss. Both examples are single addresses, but a tree node contains a range of addresses and a range lookup across nodes is supported as well.

can be realized if a huge tree is split into numerous small ones. We do so by having two layers of address mapping. In the first layer, we divide the home space into fixed-length partitions, so that a home address can be simply divided by the partition length, costing as low as one CPU cycle, to determine which partition the address locates in. In the second layer, each partition holds a small tree for further address lookup (Figure 3). Our approach can reduce the tree height by several times. With real world workloads, this optimization improves transaction throughput by 39.6% on average (§5.2).

Group update. Opportunistically merging tree nodes is another way to further reduce the number of nodes and thus the height of a tree. When two sibling nodes contain contiguous home addresses *and* map to contiguous log addresses, they can be merged. Spatially local writes within a transaction can exploit this optimization. Within each NVMM transaction, we first buffer all writes in DRAM, and combine those with contiguous home addresses on transaction commit. A group of combined writes is appended to the log and the address mapping tree is updated the minimal number of times. Overall, this optimization realizes 42.3% transaction throughput improvement according to our evaluation (§5.2).

Skip lists and locking. We choose the skip list [42], a probabilistic alternative to balanced trees, as our tree data structure (Figure 3). The main reason for our choice is that, while supporting $O(\log n)$ operations on average, the skip list does not need a complex rebalancing operation as a strictly balanced tree such as B-tree does.

Such an optimization is crucial for multi-threaded scenarios. A typical balanced tree requires a readers-writer

lock to protect concurrent operations[2]. Lock contention due to heavy reads and writes can deteriorate throughput of such systems. In contrast, by leveraging skip lists, we get rid of locking for read-only operations. Particularly, an update of the skip list involves only simple pointer manipulations on singly linked lists. Taking advantage of CPU's atomic word write (aligned 64 bits for x86), such an update is implemented in a way that is atomic to lock-free read-only operations. By avoiding such lock contention, we can see 48.9% higher transaction throughput with four threads in our experiments (§5.2).

Tree node caches. We equip each working thread with a thread-local cache that stores recently accessed home addresses and pointers to their nodes in the trees (Figure 3). When the program accesses an address, our library first searches the cache. If it is hit, the library directly gets the pointer to the tree node that contains the requested address mapping; otherwise, a full tree lookup is necessary and the resulting node is added to the cache. Such a caching mechanism is effective because of inherent temporal and spatial locality among memory accesses. As our experiments show, some memory areas are hot and frequently accessed, and memory accesses tend to cluster within 64 B areas. The hit ratio is 92.2% on average, and introduction of tree node caches leads to 30.1% increase in transaction throughput on average (§5.2).

We tweak a regular hash table design to meet a special requirement of our tree node cache. That is, once a node is cached, addresses within its mapped area tend to be a cache hit. A plain hash table does not give such a feature as cached addresses are randomly distributed. For example, a node for a 64 B area starting at 0x1000 is cached. If an access to the address 0x1008 falls into a different bucket, it would lose the chance to be checked with this node and hence be a miss. To solve the issue, we deliberately increase certain collision by using set-associativity. Based on the observation above, we try to route addresses within a 64 B scope to the same bucket so that nearby addresses can be checked with chained tree nodes that may cover them. To realize that, we pick high-order bits of an address as its hash value. Consequently, sequential addresses have a good chance of falling into one bucket.

3.3 NVMM Organization

The goal of our NVMM organization is to allow each thread to allocate NVMM with minimal overhead. Towards that end, the NVMM region is physically organized into static *chunks*, atop which we build logical logs. Multiple chunks can be linked into a list. We choose a relatively small chunk size (e.g., 32 KB), because typical NVMM writes are small; moreover, an individual chunk with a small size can be more quickly cleaned and recycled in an incremental manner.

Chunks help reduce contention among the multiple threads. We maintain a global pool of free chunks, and each thread has its own list(s) of chunks in use. A thread is allowed to buffer some free chunks when it requests one from the global pool, or after it obtains them from local log cleaning. This can avoid frequent manipulation of the global pool and its lock contention.

3.4 Log Structure

A log in the NVMM region consists of a list of chunks. Multiple logs coexist in our system. It is different from a conventional disk-based log-structure system which tends to have a single log per disk because the disk has only one disk header and sequential access is the first priority. With fast random access instead, NVMM warrants a different design, which favors thread-level parallelism by using thread-local logs. Furthermore, each thread has multiple logs to improve log cleaning efficiency, as we describe later in this section. LSNVMM employs a number of log cleaners to collect free space accumulated in chunks. The free spaces come from `pfree` operations or old data that has been updated. We use a background thread to run a cleaner.

Log entry. A log entry holds two kinds of metadata. First, a mapping for a modified or allocated memory area. When a log cleaner scans the chunk, it checks liveness of each log entry by looking up the home address from the address mapping tree. Second, a tombstone for each freed area. A tombstone is never accessed within transactions, but used on the recovery path to filter out freed areas. Atop log entries, we build transactions. A transaction consists of all log entries that it produces, by memory stores and (de)allocations.

Cleaning policy. The log cleaner moves sparse live data from several chunks to a new chunk in a compact manner, and recycles the cleaned chunks. Chunks with the amount of live data below a threshold (20% by default in our setup) are selected for cleaning.

We design three optimizations for log cleaning. **(1)** *Fast cleaning*: When all log entries in a chunk are stale, the chunk can be safely reclaimed. This can be done fast because we only need to modify a few list pointers to move the chunk to a free chunk list, without data copying. **(2)** *Separate logs*: We observe that memory stores always have better locality than memory allocations. It implies that mixing them in one log may increase the log cleaning cost and decrease the chance of fast cleaning. So we design separate logs for each thread, the *update log* serving memory stores, the *allocation log* serving memory allocations and the *deallocation log* storing only tombstones. **(3)** *Parallel cleaning*: In order to have sufficient log cleaning throughput, we

[2]There are carefully crafted lock-free balanced tree designs [8, 14] but they involve extra complexity and overhead. In contrast, our approach is simple and performs well in practice.

perform log cleaning with multiple background threads for different chunks.

4 Implementation

This section describes the implementation of LSNVMM. We start with the home space management mechanisms in §4.1, then elaborate log space management in §4.2 and address mapping between the two spaces in §4.3. Log cleaning and recovery procedures are described in §4.4 and §4.5, respectively.

4.1 Home Space Management

Memory allocation and access are two main functionalities of home space management. We draw upon existing implementation of transactional memory systems to realize such functionalities[3]. But we add persistence to transactional memory: (1) necessary allocation metadata is stored in NVMM so that the home address space can be rebuilt after a crash, and (2) committed transactions are stored in NVMM so that data updates are persistent. Next, we detail the underlying mechanisms.

Home space allocation. Considering that the 64-bit home address space is virtual and sufficiently large, fragmentation is not a severe issue there. Thus, we choose current memory allocators Hoard [6] and dlmalloc [26] to implement *home space* allocation. Hoard serves memory allocations smaller than 8 KB, while dlmalloc deals with larger ones [50].

The state of both allocators is consistently rebuilt upon crashes using metdata stored with data and therefore, no runtime effort is spent in ensuring persistence of the state. Take Hoard for example. It organizes home space into superblocks, and each superblock serves allocation requests of a certain size (e.g., a 8 KB superblock contains an array of 16 B allocations). The metadata of superblocks (location and allocation size) is stored in NVMM. With such information, we simply rely on the logs to infer allocation state after crashes. Therefore, home-space allocations do not incur any persistent operations.

Transactional memory. Applications' access to home-space data is protected by transactions. Intel STM compiler [1] is used to instrument regular C/C++ code with transaction annotations. Programmers place the keyword `__tm_atomic` and a pair of braces to specify the scope of a transaction. The compiler automatically generates calls into our transaction system when a transaction begins, issues memory loads and stores, and commits.

We employ TinySTM [16], a lightweight software transactional memory implementation, to intercept these calls and implement concurrency control of transactions. Each transaction holds a temporary private *write set* containing all written values and their addresses, which are not visible to concurrent transactions. When a transaction allocates memory, the system quickly allocates the requested size in the home space, and returns its home address. After that, all writes to the newly allocated space are buffered in the volatile write set.

Allocated memory, writes to old data are all persisted into logs when the transaction is committed. Likewise, deallocations are also logged to ensure that memory does not leak. A TinySTM transaction may receive memory writes to both volatile regions and NVMM. The LSNVMM library takes the responsibility to filter out writes to volatile memory and persist those to NVMM in a crash consistent manner when the transaction is committed. The group update optimization is performed to merge NVMM writes that have contiguous home addresses. Afterwards, a single log entry is generated for each NVMM write and flushed to logs in NVMM. Then each NVMM write obtains its log address, and the library inserts into global address mapping trees the mappings from home addresses to log addresses.

4.2 Log Space Management

From top down, the hierarchy of log storage is as follows: (1) a log is stored in a number of fixed-length *chunks*; (2) within one or more chunks, transactions that constitute the log are stored in *transaction blocks*; (3) within a transaction block, memory allocations and updates of the transaction are stored in *log entries*. We now describe these components in a bottom up order.

Log entry. Each log entry has a header and data. The header consists of (1) a 47-bit *home address* to record the start home address of the data, (2) one bit to denote whether the entry is a tombstone, and (3) a 16-bit *size* to record the data length. 47 bits are enough to hold a home address because we record the offset of the address in the NVMM region. Immediately after the header is the data whose location is its *log address*. This entry structure is used for both update and allocation logs.

Transaction block. A group of log entries belonging to a transaction make the payload of a transaction block. A preamble contains the following fields: (1) A 64-bit *version number* to record the commit time of the transaction. In our implementation, it is the monotonically increasing, globally unique timestamp generated by TinySTM for each transaction[4]. (2) A 48-bit *peer pointer* that points to another transaction block (e.g., in a different chunk as the current chunk is filled up), or in an allocation log if the current log is an update log, or vice versa. As a result, all blocks of a transaction form a

[3] LSNVMM is not bound to transactional memory. We choose the interface because it is easy to use for applications.

[4] It is an optimization to reuse the timestamp, but LSNVMM is not necessarily bound to any TM implementation. We can also simply use a global atomic counter to generate the version number.

cyclic singly-linked list. (3) A 16-bit *entry number* to record the number of log entries in the current transaction block. If the number is not enough to count all entries of a transaction, more block(s) can be linked to the current block. (4) A 32-bit *checksum* using CRC32 error-detecting code, which is calculated against the whole transaction block.

Since a logical transaction may contain multiple transaction blocks across both update and allocation logs, consistency among the blocks becomes an issue. We have to handle the issue in two cases. The first case is when a crash interrupts a transaction commit. LSNVMM can detect this case by checking the checksum of each transaction block on recovery, and discard the transaction if any of its block is invalid or lost. The second case is when a transaction block is moved to another chunk due to log cleaning. As a result, peer pointers referencing moved blocks are no longer valid. However, such inconsistency brings no problem as long as the containing transactions are safely committed, because the peer pointers are only used for detecting uncommitted transactions as in the first case. Therefore, we only need to divert log cleaning from log ends that contain uncommitted transactions.

Chunk. The payload of a chunk is a sequence of transaction blocks that make part of a log. Chunks are doubly linked by their headers. Besides, the header holds a flag to denote whether the chunk belongs to an update log or an allocation log. If a transaction block contains a log entry larger than the remaining space of a chunk, the entry can be split into more, and stored in linked peer blocks in other chunks.

4.3 Skip List

An address mapping tree is implemented as a concurrent skip list. By using insertion as an example, we show how our skip list operates in a concurrent manner. In a skip list, insertion of a node involves inserting the node to a number of levels. For each level, the insertion is identical to that of a singly linked list, which can be atomically realized by feat of atomic pointer updates. We do insertion from the bottom level up. Once the node is inserted to the bottom level, the insertion is effective. Inserting to upper levels only influences lookup performance. So, the insertion is logically atomic to concurrent reads.

While reads are lock-free, any tree structure update (e.g., insert or delete) has to hold a lock controlling the whole tree, because concurrent updates may corrupt each other. But we can still maintain high update concurrency, thanks to the large number of such trees in our design.

The tree node cache also needs a careful concurrency control. We have to check if the hit node still holds the requested home address, because it is possible the node has been removed and recycled. Accordingly, we check the home address of a node twice – before and after reading the log address of the node. If both checks match, the log address must be valid.

4.4 Log Cleaning

When memory utilization is beyond a threshold, a few background cleaner threads begin to work, in parallel with transaction threads. Cleaning steps are as follows: (1) A set of victim chunks are identified according to the policy in §3.3. For each victim chunk, a scan of all its log entries is performed to determine liveness of the data in each entry by checking its latest version in the address mapping tree, v_t. If v_t is higher than the current transaction version, the entry is discarded. (2) For a transaction block that has live entries left, the preamble is recalculated (entry number and checksum), and the entire block appended to a new chunk. (3) For the moved transaction block, a quasi TinySTM transaction is run to update global mappings with the new log addresses of the live entries. The quasi transaction is just for enforcing concurrency control. (4) After all transaction blocks are moved out of a victim chunk, the chunk is reclaimed by adding it to the global free chunk pool.

4.5 Recovery

Our recovery works in two phases to maximize thread parallelism in a manner similar to map-reduce. In the first phase, we dispatch all log chunks to the recovery threads for parallel processing. The main task of each thread is to scan the assigned chunks and group valid log entries by the partition of their home addresses. After this phase, each thread holds an array indexed by the home partition, and each element of the array has a list of log entries belonging to the partition. Note that this temporary log entry structure only contains pointers to data in NVMM and necessary metadata (version number).

In the second phase, each recovery thread takes charge of different home partitions, and the task is to replay log entries belonging to the partitions. To do so, the above lists of log entries are shuffled among threads, so that each thread holds the lists whose partitions are in the charge of the thread. Then, for each partition, the single thread in charge sorts all log entries of the partition by their home address and version number, then pick up entries with latest versions and insert their address mappings to the global address mapping tree for that partition. The approach, similar to map-reduce, avoids most thread contention.

5 Evaluation

To evaluate the performance of log-structure NVMM, we answer three questions as follows.

- *How effective are the individual optimizations we design for LSNVMM*? (§5.2)

- *How does LSNVMM perform against traditional PTM systems*? (§5.3)
- *What are the costs of log cleaning, recovery, and DRAM footprint*? (§5.4)

5.1 Experiment Setup

All the experiments are performed on a computer with 8-core Intel Xeon CPU E5-2637 v3 (3.5 GHz) and 64 GB DRAM, running 64-bit Linux kernel version 4.2.3. All results are average of five runs.

NVM simulation. As real NVMM products are not available yet, we use a simulation method akin to that in Mnemosyne [50]. We focus on effects of slow NVMM writes instead of reads, as many prior works do [35, 9, 18, 40], because the read latency of NVMM is similar to DRAM and most memory reads are effectively served by CPU caches. For a standalone NVMM write required to be immediately persisted, we introduce an extra latency. For sequential NVMM writes that are executed together, we consider both write latency and bandwidth of NVMM. The added delay is the max of the above write latency and *total write size/NVMM bandwidth*. By default, we set the write latency to 500 ns and the sustainable write bandwidth to 1 GB/s. We implement any delay by a loop reading the CPU timestamp counter (TSC) until required time has elapsed.

Benchmarks. We run five transactional benchmarks atop our systems for evaluation. The benchmarks cover both commonly used data structures and a real application: *SPS* randomly swaps elements in a large array; *RBTree*, *B+Tree* and *HashTable (HT)* perform operations on a red-black tree, a B+ tree and a hash table, respectively; *KVStore* runs a key-value store, Tokyo Cabinet [21].

For benchmarks BTree, B+Tree, HashTable and KVStore, we perform two workloads with different access patterns: the *insert* workload (Ins) inserts a number of key-value pairs, where keys are uniformly random; the *update* workload (Upd) looks up a key, and deletes it if it is found or inserts one otherwise. Keys of these pairs follow the Zipfian distribution [5, 11] so that 90% updates happen on 15% of the data. In all workloads, the value size is 128 B by default unless otherwise noted. The total number of elements/pairs in each benchmark is 10 million, resulting in 2∼4 GB of logical NVMM footprint.

5.2 Effect of Optimizations

We demonstrate the effect of *every* optimization proposed in §3.2. Comparing the library against itself provides valuable reference for other systems/implementations as such a control experiment reveals what benefit each mechanism can bring.

Evaluated systems. We add optimizations one by one to the address mapping structure, resulting in four implementations as below.

- *Base* is the baseline using a global, single skip list for whole-space address mapping.
- *2L* enhances Base with two-layer mapping. The home space is divided into 4-KB partitions, and each partition is served by a skip list for address mapping.
- *2L-GU* enhances 2L by performing group update.
- *2L-GU-C* adds thread-local tree node caches with FIFO replacement. Each cache is up to 4 M entries.

At last, we show results of LSNVMM, which is more optimized for multiple threads. 2L-GU-C uses a readers-writer lock per partition to protect a skip list from concurrency issues, while LSNVMM avoids locking for read-only operations on a skip list. So far, all optimizations are incorporated. In this experiment, we leave out log cleaning which is orthogonal to these comparisons.

Results. Figure 4 shows performance of the four implementations running the benchmarks. We make four observations. (1) 2L constantly outperforms Base for all workloads, by 39.6% on average, due to two-layer mapping. (2) 2L-GU performs 42.3% better than 2L on average, due to group update. (3) 2L-GU-C improves transaction throughputs by 30.1% on average, compared to 2L-GU, thanks to the tree node caches. (4) Overall, the above optimizations show strong performance in various benchmarks/workloads, achieving up to 268.6% (157.9% on average) performance improvement over the baseline system.

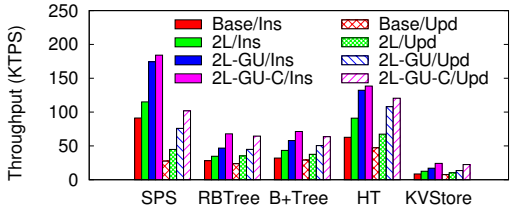


Figure 4: Transaction throughputs of the benchmarks with different optimizations, in a single thread.

Particularly, to give direct evidence of the effect of tree node caches, we plot average cache hit ratios under different benchmarks in Figure 5. A tree node cache achieves 92.2% hit ratio on average, which leads to significant performance improvement in all benchmarks.

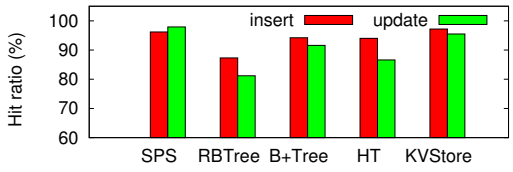


Figure 5: Access hit ratios of tree node caches under different benchmarks/workloads.

At last, our multi-thread optimization is justified by comparing 2L-GU-C and LSNVMM running the update

workload of a multi-threaded version of the data structure benchmarks, as shown in Figure 6. Removing lock overhead from read-only operations, LSNVMM achieves good scalability, and provides 48.9% higher throughput than 2L-GU-C when running four threads.

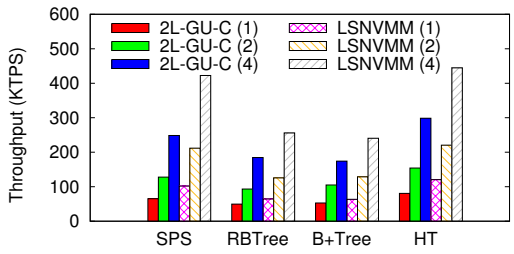


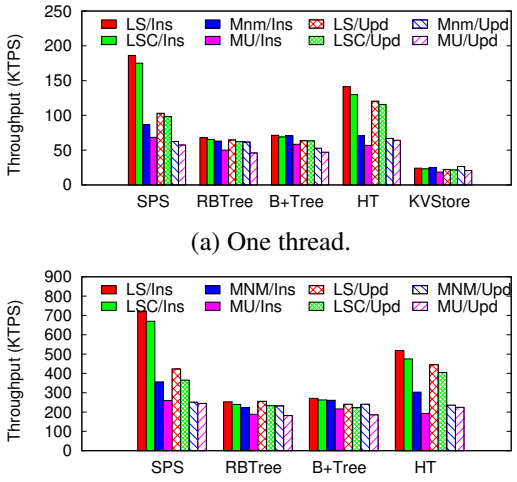
Figure 6: Multi-threaded throughputs of data structure benchmarks. "(n)" indicates the number of threads.

5.3 Comparison to Current Systems

Evaluated systems. We compare LSNVMM (LS) to redo and undo logging in the traditional memory management. In the same way as LSNVMM, both logging systems integrate with TinySTM [16]. Particularly, *Mnemosyne* (Mnm) [50] is the combination of redo logging and TinySTM with traditional memory management, and we also make *Mnmsyn-Undo* (MU) by replacing the redo logging mechanism in Mnemosyne with undo logging. Moreover, we deliberately introduce cleaning overhead to LSNVMM in *LSNVMM-Cleaning* (LSC), which triggers cleaning of chunks with over 50% stale data every around 1000 transactions.

Performance. We show performance results of the four PTM systems running the benchmarks. From Figure 7 (a), we observe that LSNVMM outperforms Mnemosyne and Mnmsyn-Undo by 37.3% and 66.1% with one thread on average, respectively. Especially for HashTable and SPS, LSNVMM achieves 89.6% and 89.9% (118.2% and 125.9%) speedup beyond Mnemosyne (Mnmsyn-Undo), respectively. These two benchmarks turn out to issue less memory loads than others. In contrast, LSNVMM does not perform well with KVStore running the update workload, mainly because it has intensive memory loads. As for scalability, Figure 7 (b) shows the performance of the PTM systems running the benchmarks in four threads. We can see that LSNVMM scales well. It performs 44.7% and 80.8% better than Mnemosyne and Mnmsyn-Undo on average, respectively. Finally, log cleaning incurs minimal overhead in this setting. Compared to LSNVMM without log cleaning, LSNVMM-Cleaning reduces the throughput of benchmarks by 4.1% and 7.8%, with one thread and four threads, respectively. More evaluation of log cleaning follows in §5.4.1.

In conclusion, LSNVMM remarkably outperforms logging PTMs even with log cleaning overhead, and shows scalability with multiple threads. LSNVMM is especially suitable for write-intensive workloads.



(a) One thread.

(b) Four threads.

Figure 7: Transaction throughputs of the benchmarks with different memory management systems.

NVMM write traffic and wear. We calculate NVMM *write traffic*, i.e., the cache line size multiplied by the total number of cache lines written back to NVMM. This metric reflects the NVMM bandwidth consumption. Among the write traffic, only modified data actually wears NVMM [57, 12], we estimate NVMM wear in terms of total *dirty bytes* ever written to NVMM. Figure 8 shows that part in breakdown of write traffic. We make two observations. (1) LSNVMM saves 82.8% and 82.0% write traffic of Mnemosyne and Mnmsyn-Undo on average, respectively. Besides the fact that redo/undo logging logically writes twice what LSNVMM does, we can clearly see the influence of cache line granularity. As home-space updates in Mnemosyne and Mnmsyn-Undo are typically sparse and fine-grained, they waste lots of NVMM traffic on flushing *entire* cache lines. (2) LSNVMM reduces dirty bytes by 80.1% and 65.1% compared to Mnemosyne and Mnmsyn-Undo on average, respectively. Thanks to the group update technique, LSNVMM merges a large number of sequential and repeated writes. On the contrary, Mnemosyne persists every write of the transaction in the log, even if it can be merged or coalesced with others.

NVMM fragmentation. In this experiment, we test different memory allocators under three typical workloads [46] that emulate variation of data value sizes. All workloads consist of two phases with different individual allocation sizes. W1 first allocates collectively 1 GB in randomly 100 - 150 bytes, and then repeats so in randomly 200 - 250 bytes. W2 is different with W1 only in that it frees 90% of the memory allocated in the first

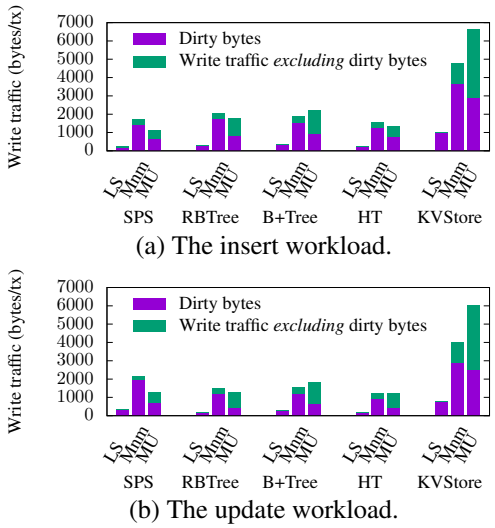


Figure 8: NVMM write traffic and wear of different memory management systems running the benchmarks (in a single thread).

phase before it goes to the second phase. W3 has the same behavior as W2 except that its individual allocation size in the first phase is random 1,000 - 2,000 bytes and in the second phase random 1,500 - 2,500 bytes.

Figure 9 depicts the results. We make two observations. (1) Typical DRAM-oriented memory allocators hardly manage memory efficiently in these workloads. Mnemosyne (Hoard) produces 25.3% memory fragmentation on average, and NVML (jemalloc) produces 35.0%. In contrast, LSNVMM keeps it as low as 4.5% by virtue of log cleaning. (2) The memory fragmentation of LSNVMM is inversely proportional to the allocation size, because each allocation has its own *metadata* cost. For example, LSNVMM incurs 7.3% fragmentation in W1 but only 0.6% in W3.

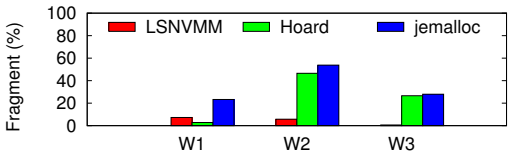


Figure 9: NVMM fragmentation ratios of LSNVMM and two other representative traditional memory allocators, Hoard [6] adopted by Mnemosyne [50], and jemalloc [15] adopted by Intel's NVML [22].

5.4 Log-Induced Costs

5.4.1 Log Cleaning

We first evaluate the effect of separate logs on fast cleaning (§3.4). Figure 10 depicts the amount of log data that is reclaimed by fast cleaning as the number of update operations increases. In the experiment, we firstly insert 10 million elements to the corresponding benchmarks. We make two observations from this figure. (1) Beyond initial 10 million updates, the fast cleaning can effectively clean around more than 200 MB memory per million updates. (2) The separate log design can clean more chunks than the baseline. Their gap is bigger in the RBTree benchmark, because it has more clustered memory stores than HashTable so that a separate update log is apt to fast cleaning.

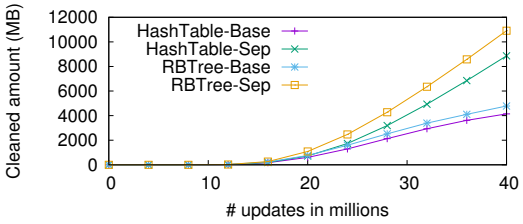


Figure 10: Fast cleaning performance with baseline ("Base") or separate logs ("Sep") with a random 1 KB update workload on HashTable or RBTree.

When all cleaning overhead walks in, Figure 11 shows the resulting performance of the benchmark as well as the throughput of the cleaner. In the experiment, we preload B+Tree to occupy a certain fraction of NVMM, and then run the update workload with four working threads and two cleaning threads. We test two cases where the value size is 128 B and 1 KB, respectively. We draw a major conclusion from this figure: LSNVMM does not lose much performance under high NVMM pressure. The performance degradation due to cleaning was 8% or less, even at 90% memory utilization.

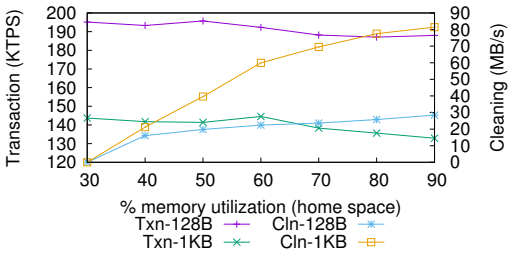


Figure 11: Transaction ("txn") and cleaning ("cln") throughputs of the B+Tree benchmark with random key distribution and different value sizes (128B vs. 1KB) as a function of memory utilization.

5.4.2 Recovery

Figure 12 shows the required time to recover from a 10 GB of logs in NVMM. We rebuild the whole LSNVMM in multiple threads. We make two observations from this figure: (1) The recovery process quickly speeds up with more threads. For 128 B values, LSNVMM needs 19.2 seconds to recover in one thread,

but only 3.0 seconds in eight threads. (2) The recovery latency is inversely proportional to the data allocation size, because the number of address mappings decreases as the allocation size increases.

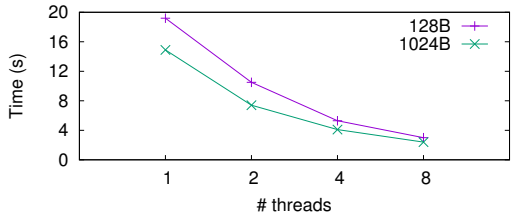


Figure 12: Recovery time of 10-GB NVMM logs, with different numbers of threads and different value sizes (128B vs. 1KB).

5.4.3 DRAM footprint

We evaluate the DRAM footprints using the real application KVStore under the insert workload with different value sizes. Figure 13 illustrates the amount of DRAM required. It is around 16.9% of NVMM when the value size is as small as 128 B, and drops quickly as the value size increases.

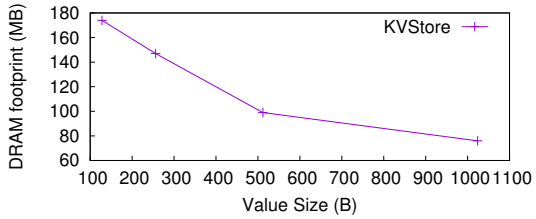


Figure 13: DRAM footprint of the address mapping structures and thread cache in KVStore for 1 GB NVMM data as a function of the value size.

6 Related Work

Persistent memory systems. They can be classified into three categories by their interfaces. One category is PTM. For example, Mnemosyne [50], SoftWrAP [18] and DudeTM [33] are redo logging based PTMs, while NV-Heaps [10], NVML [22] and DCT [24] are undo logging based ones. Our work is built on many PTM techniques, but follows a different, log-structured way to address the memory management issue.

The second category provides data structure interfaces, such as CDDS [49] and NV-Tree [53]. Their interfaces to applications are not as flexible as transactions. The third category is software transparent. WSP [37] and ThyNVM [44] are two representatives. They either have a strong assumption on hardware or involve advanced hardware features. In contrast, LSNVMM is a general solution and requires no customized hardware.

Memory allocators. Makalu [7] and nvm_malloc [47] are NVMM allocators that aim at collecting garbage in a failure-safe manner. WAlloc [54] proposes a wear-aware memory allocator to improve the wear leveling. These works address other aspects of memory management, while we focus on the memory fragmentation problem.

RAMCloud [46] shares the same goal as our work to reduce memory fragmentation. It also uses a log-structured approach. But it is a key-value store of well-defined data objects, without the need for a tree-based address mapping mechanism as in LSNVMM. LSNVMM supports general transactions for arbitrary data.

Log-structured systems. The log-structure approach was early designed in LFS [45], which buffers random writes in DRAM and makes best use of sequential I/O of hard disk drives. F2FS [29] proposes a well optimized file system on flash storage devices, which adopts separate metadata and data logs, and uses adaptive logging to avoid frequent garbage collection. It is similar to our separate log design. NOVA [52] is a file system optimized for hybrid memory systems, providing strong consistency guarantees. It maintains independent logs for each inode to improve scalability. Some databases [4] implement log-structured data management, and take advantage of NVMM to simplify traditional DBMS. Overall, the log-structured approach is widely used in those systems, but their designs hardly apply to LSNVMM whose unique challenge is tree-based address translation as discussed in §2.

7 Conclusion

The log-structured NVMM eliminates the dichotomy between data home and data logs in current logging PTMs. This solves the vital NVMM fragmentation issue, and lowers NVMM write wear and persistence overhead. To that end, we create four key optimizations to tackle the performance challenge in tree-based address mapping. Our experiments show that the log-structured NVMM can outperform Mnemosyne and Mnmsyn-Undo by 44.7% and 80.8% on average in terms of transaction throughput. Our work reveals how a software tree structure can be optimized to a level that can efficiently serve address mapping for NVMM load/store instructions.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Yu Hua, for their valuable feedback. This work was partially supported by the National Natural Science Foundation of China (Grant No. 61502266, 61433008, 61232003), the Beijing Municipal Science and Technology Commission of China (Grant No. D151100000815003), and the China Postdoctoral Science Foundation (Grant No. 2016T90094, 2015M580098).

References

[1] Intel C++ STM compiler prototype edition. https://software.intel.com/en-us/forums/intel-c-stm-compiler-prototype-edition, 2012.

[2] AKINAGA, H., AND SHIMA, H. Resistive random access memory (ReRAM) based on metal oxides. *Proc. IEEE 98*, 12 (2010).

[3] APALKOV, D., KHVALKOVSKIY, A., WATTS, S., NIKITIN, V., TANG, X., LOTTIS, D., MOON, K., LUO, X., CHEN, E., ONG, A., DRISKILL-SMITH, A., AND KROUNBI, M. Spin-transfer torque magnetic random access memory (STT-MRAM). *ACM J. Emerg. Technol. Comput. Syst. 9*, 2 (May 2013), 13:1–13:35.

[4] 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* (2015), SIGMOD '15, pp. 707–722.

[5] 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), SIGMETRICS '12, pp. 53–64.

[6] 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* (2000), ASPLOS IX, pp. 117–128.

[7] BHANDARI, K., CHAKRABARTI, D. R., AND BOEHM, H.-J. Makalu: Fast recoverable allocation of non-volatile memory. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2016), OOPSLA '16, pp. 677–694.

[8] BRAGINSKY, A., AND PETRANK, E. A lock-free B+tree. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures* (2012), SPAA '12, pp. 58–67.

[9] CHATZISTERGIOU, A., CINTRA, M., AND VIGLAS, S. D. REWIND: Recovery Write-ahead system for In-memory Non-volatile Data-structures. *Proc. VLDB Endow. 8*, 5 (Jan. 2015), 497–508.

[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), ASPLOS XVI, 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), SoCC '10, pp. 143–154.

[12] DU, Y., ZHOU, M., CHILDERS, B. R., MOSSÉ, D., AND MELHEM, R. Bit mapping for balanced PCM cell programming. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (2013), ISCA '13, pp. 428–439.

[13] 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* (2014), EuroSys '14, pp. 15:1–15:15.

[14] ELLEN, F., FATOUROU, P., HELGA, J., AND RUPPERT, E. The amortized complexity of non-blocking binary search trees. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing* (2014), PODC '14, pp. 332–340.

[15] EVANS, J. A scalable concurrent malloc(3) implementation for FreeBSD. In *Proceedings of the BSDCan Conference* (2006).

[16] FELBER, P., FETZER, C., AND RIEGEL, T. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2008), PPoPP '08, pp. 237–246.

[17] FREE SOFTWARE FOUNDATION, INC. *The GNU C Library*, 2.24 ed., Aug. 2016. https://www.gnu.org/software/libc/manual/.

[18] GILES, E. R., DOSHI, K., AND VARMAN, P. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *Proceedings of the 31st Symposium on Mass Storage Systems and Technologies* (May 2015), MSST '15, pp. 1–14.

[19] GUO, F., AND EFSTATHOPOULOS, P. Building a high-performance deduplication system. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (2011), USENIX ATC '11, pp. 271–284.

[20] HERTZ, M., AND BERGER, E. D. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (2005), OOPSLA '05, pp. 313–326.

[21] HIRABAYASHI, M. Tokyo cabinet: a modern implementation of DBM. `http://1978th.net/tokyocabinet/`, 2010.

[22] INTEL. The NVM Library. `http://pmem.io/`, 2016.

[23] INTEL NEWSROOM. Introducing Intel Optane technology – bringing 3D XPoint memory to storage and memory products. `https://newsroom.intel.com/press-kits/introducing-intel-optane-technology-bringing-3d-xpoint-memory-to-storage-and-memory-products/`, July 2015.

[24] KOLLI, A., PELLEY, S., SAIDI, A., CHEN, P. M., AND WENISCH, T. F. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (2016), ASPLOS '16, pp. 399–411.

[25] KLTRSAY, E., KANDEMIR, M., SIVASUBRAMANIAM, A., AND MUTLU, O. Evaluating STT-RAM as an energy-efficient main memory alternative. In *Proceeding of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software* (Apr. 2013), ISPASS '13, pp. 256–267.

[26] LEA, D. A memory allocator. `http://g.oswego.edu/dl/html/malloc.html`.

[27] LEE, B., ZHOU, P., YANG, J., ZHANG, Y., ZHAO, B., IPEK, E., MUTLU, O., AND BURGER, D. Phase-change technology and the future of main memory. *IEEE Micro 30* (Jan. 2010), 131–141.

[28] LEE, B. C., IPEK, E., MUTLU, O., AND BURGER, D. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (2009), ISCA '09, pp. 2–13.

[29] LEE, C., SIM, D., HWANG, J., AND CHO, S. F2FS: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies* (2015), FAST '15, pp. 273–286.

[30] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZISE, G., AND CAMBLE, P. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proccedings of the 7th Conference on File and Storage Technologies* (2009), FAST '09, pp. 111–123.

[31] 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), SOSP '11, pp. 1–13.

[32] LINUX KERNEL ORGANIZATION, INC. Direct access for files. `https://www.kernel.org/doc/Documentation/filesystems/dax.txt`, 2016.

[33] LIU, M., ZHANG, M., CHEN, K., QIAN, X., WU, Y., AND REN, J. DudeTM: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (2017), ASPLOS '17, pp. 329–343.

[34] LIU, R.-S., SHEN, D.-Y., YANG, C.-L., YU, S.-C., AND WANG, C.-Y. M. NVM Duet: Unified working memory and persistent store architecture. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (2014), ASPLOS '14, pp. 455–470.

[35] LU, Y., SHU, J., AND SUN, L. Blurred persistence in transactional persistent memory. In *Proceedings of the 31st Symposium on Mass Storage Systems and Technologies* (May 2015), MSST '15, pp. 1–13.

[36] MORARU, I., ANDERSEN, D. G., KAMINSKY, M., TOLIA, N., RANGANATHAN, P., AND BINKERT, N. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems* (2013), TRIOS '13, pp. 1:1–1:17.

[37] NARAYANAN, D., AND HODSON, O. Whole-system persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), ASPLOS XVII, pp. 401–410.

[38] 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), NSDI '13, pp. 385–398.

[39] ORACLE. NVM Direct. `https://github.com/oracle/nvm-direct`, 2016.

[40] OU, J., SHU, J., AND LU, Y. A high performance file system for non-volatile main memory. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), EuroSys '16, pp. 12:1–12:16.

[41] PELLEY, S., CHEN, P. M., AND WENISCH, T. F. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (2014), ISCA '14, pp. 265–276.

[42] PUGH, W. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM 33*, 6 (June 1990), 668–676.

[43] RAOUX, S., BURR, G. W., BREITWISCH, M. J., RETTNER, C. T., CHEN, Y.-C., SHELBY, R. M., SALINGA, M., KREBS, D., CHEN, S.-H., LUNG, H.-L., AND LAM, C. H. Phase-change random access memory: A scalable technology. *IBM J. Res. Dev. 52*, 4 (July 2008), 465–479.

[44] REN, J., ZHAO, J., KHAN, S., CHOI, J., WU, Y., AND MUTLU, O. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th International Symposium on Microarchitecture* (2015), MICRO-48, pp. 672–685. `http://persper.com/thynvm/`.

[45] 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.

[46] 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* (2014), FAST '14, pp. 1–16.

[47] SCHWALB, D., BERNING, T., FAUST, M., DRESELER, M., AND PLATTNER, H. nvm_malloc: memory allocation for NVRAM. In *In Sixth International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (in conjunction with VLDB)* (2015).

[48] SEARS, R., AND RAMAKRISHNAN, R. bLSM: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), SIGMOD '12, pp. 217–228.

[49] 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* (2011), FAST '11, pp. 61–75.

[50] 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* (2011), ASPLOS XVI, pp. 91–104.

[51] WANG, C., CHEN, W.-Y., WU, Y., SAHA, B., AND ADL-TABATABAI, A.-R. Code generation and optimization for transactional memory constructs in an unmanaged language. In *Proceedings of the International Symposium on Code Generation and Optimization* (2007), CGO '07, pp. 34–48.

[52] XU, J., AND SWANSON, S. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies* (2016), FAST '16, pp. 323–338.

[53] 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), FAST '15, pp. 167–181.

[54] YU, S., XIAO, N., DENG, M., XING, Y., LIU, F., CAI, Z., AND CHEN, W. Walloc: An efficient wear-aware allocator for non-volatile main memory. In *2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC)* (2015), IPCCC '15, pp. 1–8.

[55] ZHANG, Y., AND SWANSON, S. A study of application performance with non-volatile main memory. In *Proceedings of the 31st Symposium on Mass Storage Systems and Technologies* (May 2015), MSST '15, pp. 1–10.

[56] 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* (2013), MICRO-46, pp. 421–432.

[57] ZHOU, P., ZHAO, B., YANG, J., AND ZHANG, Y. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (2009), ISCA '09, pp. 14–23.

Soft Updates Made Simple and Fast on Non-volatile Memory

Mingkai Dong, Haibo Chen
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

Abstract

Fast, byte-addressable NVM promises near cache latency and near memory bus throughput for file system operations. However, unanticipated cache line eviction may lead to disordered metadata update and thus existing NVM file systems (NVMFS) use synchronous cache flushes to ensure consistency, which extends critical path latency.

In this paper, we revisit soft updates, an intriguing idea that eliminates most synchronous metadata updates through delayed writes and dependency tracking, in the context of NVMFS. We show that on one hand byte-addressability of NVM significantly simplifies dependency tracking and enforcement by allowing better directory organization and closely matching the per-pointer dependency tracking of soft updates. On the other hand, per-cache-line failure atomicity of NVM cannot ensure the correctness of soft updates, which relies on block write atomicity; page cache, which is necessary for *dual views* in soft updates, becomes inefficient due to double writes and duplicated metadata. To guarantee the correctness and consistency without synchronous cache flushes and page cache, we propose pointer-based *dual views*, which shares most data structures but uses different pointers in different views, to allow delayed persistency and eliminate file system checking after a crash. In this way, our system, namely SoupFS[1], significantly shortens the critical path latency by delaying almost all synchronous cache flushes. We have implemented SoupFS as a POSIX-compliant file system for Linux and evaluated it against state-of-the-art NVMFS like PMFS and NOVA. Performance results show that SoupFS can have notably lower latency and modestly higher throughput compared to existing NVMFS.

1. Introduction

Soft updates, which uses delayed writes for metadata updates, tracks per-pointer dependencies among updates in memory, and enforces such dependencies during write back to disk, is an intriguing idea that promises metadata update latency and throughput close to memory-only file systems [10, 11, 23, 35]. However, soft updates is also known for its high complexity, especially the complex dependency tracking as well as enforcement (like rollback/forward to resolve cyclic dependencies, which also lead to double writes) [1, 3, 9, 13, 22]. A known file system developer Valerie Aurora argued that "soft updates are, simply put, too hard to understand, implement, and maintain to be part of the mainstream of file system development" [1].

In this paper, we revisit soft updates for NVM and argue that two main sources of the complexity are: 1) *the mismatch between per-pointer based dependency tracking and the block-based interface of traditional disks*; 2) *excessively delayed writes that complicate dependency tracking*. We then show that soft updates can be made simple by taking advantage of the byte-addressability and low latency offered by NVM. Byte-addressability matches the per-pointer based dependency tracking by eliminating false sharing among different data structures and avoiding cyclic dependencies and complex rollback/forward. Byte-addressability also allows the use of more efficient data structures like hash tables in the directory organization to further simplify the dependencies of file system operations. Besides, page cache and disk scheduler can be excluded from the storage hierarchy because of the byte-addressability and low latency of NVM, so that soft updates can use in-place writes with delayed persistency to simplify the dependency tracking. The simplified storage hierarchy also eliminates the gap between the page cache and the file system, making the dependency tracking and enforcement semantic-aware and even simpler.

However, there is still a major challenge that impedes the application of soft updates to NVM. Since page cache, the software cache layer designed for slow storage media, is removed for performance concerns, file system updates are directly written to CPU cache and persisted to NVM later. Unlike page cache that can be precisely controlled by file systems, CPU cache is hardware-managed such that file systems cannot control the eviction of cache lines. State-of-the-art NVM file systems (NVMFS) [8, 45], like traditional disk-based file systems, use logging or shadow copying to ensure crash consistency. Yet, instead of buffering data for explicit and periodic flushing later, NVMFS has to eagerly flush critical metadata in case of accidental eviction of such metadata to NVM in a wrong order. This necessitates the uses of high latency operations like *clflush/clflushopt+sfence* in

[1] Short for Soft updates inspired File System

the critical path of file system related syscalls, which inevitably extends the critical path latency.

To overcome the consistency issue from unanticipated cache line eviction without page cache and cache flush operations, we review *dual views*, a latest view and a consistent view, which is used in soft updates for file system metadata. All metadata in the consistent view is always persisted and consistent, while metadata in the latest view is always up-to-date and might be volatile. Without caring about the cache line eviction, a syscall handler operates directly in the latest view and tracks the dependencies of modifications. Unanticipated cache line eviction in the latest view can never affect the persisted metadata in the consistent view by design. Background *persisters* are responsible for asynchronously persisting metadata from the latest view to the consistent view according to the tracked dependencies. They use *clflush/clflushopt+sfence* operations to enforce the update dependencies in background without affecting the syscall latency. A naive approach to providing *dual views* is duplicating all metadata in the file system. Such an approach doubles the memory usage and causes unnecessary memory copies when synchronizing metadata between the latest view and the consistent view. To implement *dual views* efficiently, we propose pointer-based *dual views*, in which most structures are shared by both views and different views are observed by following different pointers. Thanks to pointer-based *dual views*, SoupFS avoids almost all synchronous cache flushes in the critical path, and the consistent view can be immediately used without performing file system checking or recovery after crashes.

We have implemented SoupFS as a POSIX-compliant[2], NVM-based file system at the backend of the virtual file system in Linux. Evaluations using different NVM configurations show that SoupFS provides notably lower latency and modestly higher throughput compared to state-of-the-art NVM file systems such as PMFS and NOVA. Specifically, SoupFS achieves up to 80% latency reduction for file system related syscalls in the micro-benchmarks and improves the throughput by up to 89% and 50% for Filebench and Postmark.

In summary, the contributions of this paper include:

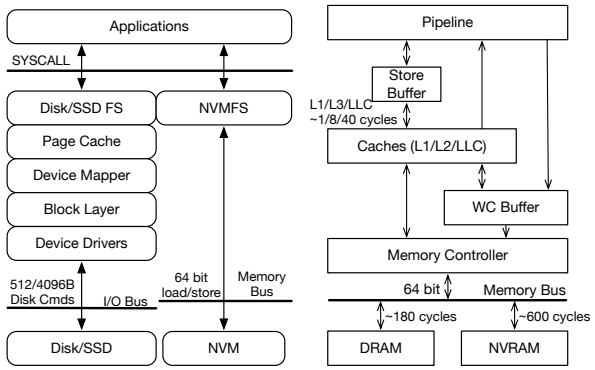
- A detailed analysis of the complexity of soft updates and the argument that soft updates can be made simple for NVM (§2).
- A review of the update dependencies of file systems on NVM, a simple semantic-aware dependency tracking and enforcement mechanism and efficient pointer-based *dual views* (§3).
- An implementation of SoupFS on Linux and an extensive evaluation (§4) that confirms the efficiency of SoupFS.

[2] SoupFS has passed the POSIX-compliant test in `http://www.tuxera.com/community/posix-test-suite/`.

2. Background and Motivation

2.1 NVM and NVMFS

Emerging NVM technologies such as PCM, STT-MRAM, Memristor, NVDIMM and Intel/Micron's 3D XPoint are revolutionizing the storage hierarchy by offering features like byte-addressability, non-volatility, and close-to-DRAM speed. STT-RAM has lower latency than DRAM but high cost, making it a promising replacement for on-chip cache instead of DRAM replacement [47]. Other emerging NVM media like PCM or 3D XPoint generally have higher latency especially higher write latency than DRAM, which indicates that synchronous write to NVM would cause higher latency than that to DRAM. NVDIMM, a commercially available NVM solution, generally has the same performance characteristics with DRAM as it is essentially battery-backed DRAM, though it is usually with 10–20X higher price than DRAM according to a recent price quotation.



(a) Storage stack (b) CPU path

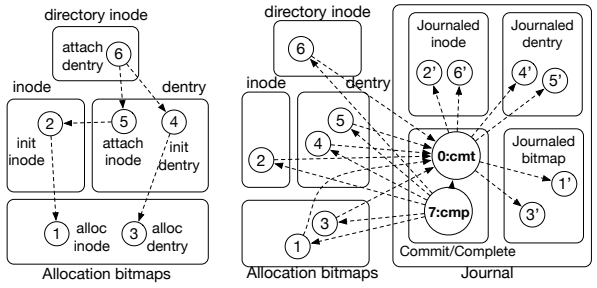
Figure 1: Storage stack and path

While new software can leverage the load/store interface to access NVM directly, quite a lot of software may continue to access persistent data in NVM through the file system interface. Hence, there have been intensive efforts in designing NVM file systems (NVMFS) [4–6, 8, 25, 44, 45]. Figure 1(a) illustrates the storage stacks from applications to persistent storage for disks (including SSD) and NVM. Compared to disk file systems, NVMFS can avoid major storage software layers like page cache, device mapper, block layer and drivers, but instead only relies on memory management for space management. However, there are also several challenges that require a redesign of file systems for NVM.

Fine-grained Failure-Atomic Updates: Although it is claimed that memory controllers supporting Intel DIMM will also support Asynchronous DRAM Refresh [32], the failure-atomic write unit is only one cache line size, still far less than 512-/4096-byte for disks. This fine-grained failure atomicity prevents the use of prior approaches (like backpointers [3]) relying on coarse-grained failure atomicity.

Hardware-controlled Ordering: NVMFS elevates the level of persistency boundary from DRAM/Disk to CPU cache/NVM. However, unlike disk-based file systems that have complete control of the order of data flushed to disk, CPU cache is hardware-managed and unanticipated cache line eviction may break the ordering enforced by sfence/mfence, which only orders on-chip visibility of data updates across CPU cores. To this end, prior NVMFS needs to eagerly use *clflush* or *clflushopt* to flush data from CPU cache to the memory controller [4, 8, 25, 44, 45]. *clflushopt* allows asynchronously flushing data compared to synchronous and serialized feature of *clflush*. But the ordering of *clflushopt* must be enforced by memory fences like *sfence*. Eagerly flushing cache lines and persisting data would cause high latency in the critical path, especially for NVM with higher write latency than DRAM.

Software Efficiency: Unlike in conventional file systems where slow storage devices dominate access latency, the cost of *clflush* and *clflushopt+sfence* is much higher compared to CPU cache accesses. It becomes the new bottleneck and must be minimized in the critical path to approach the near-cache access speed. Besides, the scarcity of CPU cache volume requires economizing cache usage to provide more space for applications.



(a) General (b) Journaling

Figure 2: Persistency dependency of creating a file

2.2 The Cost of Consistency

To address fine-grained atomicity and hardware-controlled ordering, prior file systems need to order operations carefully and use synchronous flushing to preserve crash consistency, which leads to high latency. Figure 2(a) illustrates the dependency to create a file, where the dashed arrows denote the persistency ordering. For example, the arrow from *init inode* to *alloc inode* dictates that the initialization of the new inode must not be persisted until its allocation information is made persistent. Prior file systems like PMFS [8] usually use journaling to handle this issue, which further complicates the dependencies (as shown in Figure 2(b)) and still requires eagerly flushing the logs. In this example, there are around 19 persistency dependencies to be respected, which requires around 14 *clflushes*. Packing multiple journaled metadata into a single cache line can reduce the number of *clflushes*, but cannot eliminate them.

2.3 Soft Updates

Soft updates [10, 11, 23, 35] is an intriguing metadata update technique and has been implemented in FreeBSD UFS [23]. It has the promise of mostly eliminating synchronous metadata updates and providing fast crash recovery by instantly providing a consistent file system. While soft updates is an intriguing and concise idea to achieve low latency and high throughput, the block interface exposed by the underlying disk complicates dependency tracking and enforcement in the following ways:

Block-oriented directory organization complicates dependencies: Like many other disk-based file systems, soft updates treats directories as regular files organized by direct blocks and indirect blocks. This block-oriented directory organization simplifies the implementation of file systems for block devices but complicates the dependencies due to false sharing. For example, placing multiple dentries in the same block allows cyclic dependencies, which must be resolved by complicated roll-back/forward. It also necessitates the additional tracking of whether the block to store the new dentry is newly allocated or reused, so that it can be treated differently in the enforcement.

Delayed writes complicate dependency tracking: Delaying disk writes of metadata updates is one key idea of soft updates specially designed for disk-based storage with high write latency. A sequence of dependent metadata changes, which otherwise can be written synchronously, is delayed with various dependency tracking structures attached. While asynchronous disk writes improve creation throughput by a factor of two compared with synchronous writes [23], soft updates must track the status of delayed operations to maintain ordering for integrity and security. However, the page cache usually is unaware of the content in the page, which creates a semantic gap between the page cache (where enforcement happens) and the file system (where tracking happens). The gap forces soft updates to involve complex structures for status and dependency tracking, which complicates both the critical path of synchronous system calls and the *syncer* daemon that is responsible for flushing the delayed writes.

Roll-back/forward complicates dependency enforcement: Soft updates tracks per-pointer metadata updates to eliminate false sharing. However, during enforcement, as a disk block still contains many metadata structures, there are still many cyclic dependencies at the block level during write-back. Soft updates handles this complexity by rolling back metadata changes that have pending dependencies to only write consistent metadata updates and then rolling forward the reverted metadata changes to persist the change again. This, however, would double the disk updates and diminish its gain over journaling mechanisms [36].

Soft updates is considered difficult and complicated to implement and maintain [1, 13]. By rethinking soft updates on NVM, we find that the byte-addressability of NVM can simplify the design of soft updates and delayed persistency of soft updates can further boost the performance of file systems on NVM.

3. Design and Implementation

To embrace high performance and byte-addressability of NVM, we design SoupFS, a soft updates implementation that is simple and fast on NVM. SoupFS redesigns the directory organization using hash tables to simplify the complicated dependencies caused by block-oriented directory organization. The roll-back/forward complexity is eliminated by removing page cache, thanks to byte-addressability of NVM. The removal of page cache also enables a semantic-aware dependency tracking which alleviates the complexity caused by delayed writes.

As a result, a syscall handler simply tracks the operation type along with related pointers, and with file system semantics in mind, background *persisters* can enforce the persistency and dependencies according to the type and pointers tracked during the syscall.

SoupFS is fast mainly because it applies delayed persistency which eliminates almost all synchronous cache flushes in the file system syscall critical path. Providing *dual views*, a latest view and a consistent view, of file system metadata is the key technique to allow delayed persistency and eliminate file system checking after a crash.

However, page cache, which facilitates the implementation of *dual views* in soft updates, is removed in SoupFS for performance and simplicity. To provide *dual views* without page cache, we propose efficient pointer-based *dual views* by specially designing its metadata structures so that most structures are shared by both views and different views are observed by following different pointers.

3.1 Directory Organization

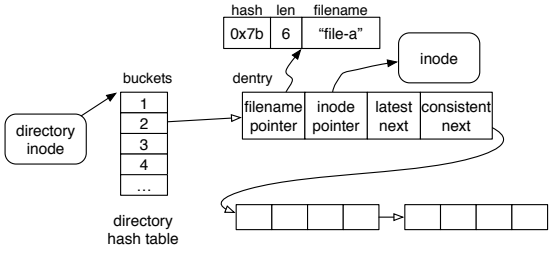


Figure 3: Directory and dentries in SoupFS

Directory operations are the core of many file system syscalls. In traditional file systems, a directory is organized akin to regular files but with different registered operations. Despite poor performance of lookups due to linear scans, reusing the regular file structures as dentry arrays is simple to implement and conforms to the usage of block devices. However, storing multiple variable-length dentries in one block causes false sharing that allows cyclic dependencies, which must be resolved by roll-back/forward and thus significantly complicates the dependency enforcement.

With the byte-addressability of NVM, we re-organize directories using hash tables as shown in Figure 3. The root of a directory points to an array of buckets, each of which points to a list of dentries. A dentry is a fixed-sized structure consisting of four pointers to the filename, the inode, the latest and the consistent next dentry. The filename is externalized to reduce fragmentation, and the hash value and length are stored ahead of the filename for fast comparison. Next pointers point to the next dentry in the hash table. Two next pointers are stored since a dentry can be in two hash tables (*dual views* of the directory) at the same time. The usage of these two next pointers is explained in §3.4.

Hash-table-based directory organization simplifies dependencies in SoupFS. Finer-grained structures used in hash tables avoid false sharing and further the roll-back/forward in the enforcement. Also, since SoupFS allocates a new dentry for each directory insertion, we don't need to track the dependency additionally. As a result, tracking the operation type and a pointer to the added/removed dentry is sufficient for persisting the metadata and enforcing the dependencies for most of the time. Update dependencies are further discussed in §3.3.

A dentry occupies 32B which is usually less than one cache line size. In the implementation, the first few bytes of a filename can be stored together with its corresponding dentry for memory efficiency.

3.2 Content Oblivious Allocator

Some file systems like EXT4 pre-allocate dedicated space for inodes. Dedicating space for inodes facilitates inode management and yields good performance in disk-based file systems. However, it fixes the number of available inodes and incapacitates the file system when inode area is full even though free data blocks are abundant. Such an issue is exacerbated significantly when more data structures are involved, such as the filename and the dentry in SoupFS.

To address this issue, we provide a content-oblivious allocator which treats the whole NVM space as a large memory pool and allocates memory without knowing what the memory is used for. The content-unawareness of the allocator breaks the boundary between various data structures, making the memory management more flexible and simpler without sacrificing performance and correctness.

We also categorize the data structures into two kinds according to the size they use for allocation (see Table 1). As a result, the content-oblivious allocator only needs to manage the memory in page size (4KB) and cache line

Table 1: Data structure sizes in SoupFS

Data Structure	Size	Allocation Size
inode	64B	64B
filename	variable	64B
dentry	32B	64B
hash table (buckets)	4KB	4KB
B-tree node	4KB	4KB
data block	4KB	4KB

size (64B). Filenames, the only variable-length structure, are split into multiple cache lines linked with pointers if a single cache line is not sufficient (see Figure 4).

Metadata of the allocator is stored in the bitmap in NVM, and in-DRAM per-CPU free-lists are used to improve the performance of frequent allocations and deallocations. The implementation of the allocator is derived from ssmalloc [21] and simplified according to two fixed allocation sizes.

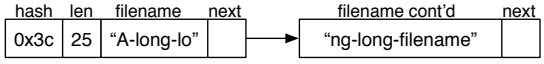


Figure 4: Long filenames in SoupFS

3.3 Update Dependencies

It is seldom a single operation to finish a syscall in file systems. Different data structures are modified in the file system, and the orders of these modifications are dedicatedly arranged for crash consistency. Soft updates summaries these ordering requirements in three rules:

- *C1:* Never point to a structure before it has been initialized, e.g., an inode must be initialized before a directory entry references it.
- *C2:* Never re-use a resource before nullifying all previous pointers to it, e.g., an inode's pointer to a data block must be nullified before that disk block may be re-allocated for a new inode.
- *C3:* Never reset the old pointer to a live resource before the new pointer has been set, e.g., when renaming a file, do not remove the old name for an inode until after the new name has been written.

These three rules are the guidelines of soft updates dependency tracking and enforcement. SoupFS follows *C2* and *C3* and generalizes *C1* which is over-restrictive in most file system operations. Taking the file creation as an example, the new dentry can hold the reference to the new inode even before the initialization of the inode is persisted, as long as the dentry has not been persistently inserted into the hash table in NVM. That is, before the dentry becomes reachable from the root, pointers in the dentry can temporarily violate *C1* without causing any consistency issue. Based on such an observation, we generalize *C1* to be "never let the structure be reachable from the root until it has been initialized," which can further simplify the dependencies in SoupFS.

We then review the update dependencies in different file system operations in SoupFS. For a file creation, a series of operations need to be done as shown in Figure 5.

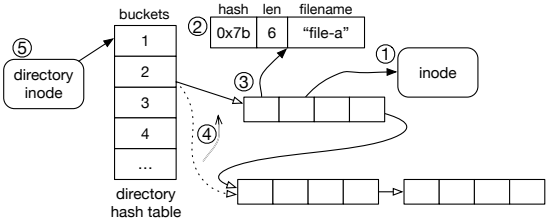


Figure 5: Dependencies of creating a file

① An inode is allocated and initialized with correct information. ② A block of memory is allocated and initialized with the filename. ③ A dentry is allocated and the pointers to the inode and filename are filled. ④ The dentry is inserted into the hash table of the directory. ⑤ The inode of the parent directory is updated. There are several pointers in the above operations. However, the only persistency dependencies we need to guarantee are:

1. *① ② ③ are persisted before the insertion of the dentry is persisted (④).*
2. *The parent directory inode information(⑤) is persisted after the persistence of dentry insertion (④).*

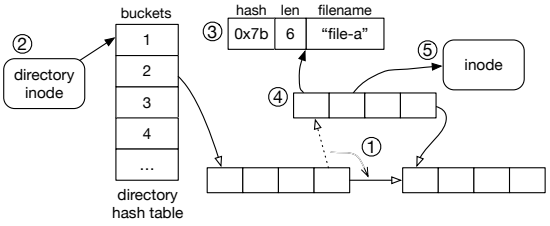


Figure 6: Dependencies of removing a file

For a file removal, the operations are reverted as shown in Figure 6[3]. ① Remove the dentry from the hash table. ② The parent directory inode is modified. ③ The filename can be freed. ④ The dentry can be freed. ⑤ The inode can be freed if its link count is zero. This time, the only ordering requirement is that *② ③ ④ ⑤ shall be done after the dentry removal (①) is persisted.*

The creation and removal of directories are largely as described above. One difference is that additional operations for hash table construction and destruction are needed. The construction of a hash table includes allocating a page as hash table buckets and erasing it to be all zeros. The destruction is simply freeing the memory since a directory can be removed only if it is empty, i.e. there is no dentry in the hash table. Additionally, we omit the "." dentry since it points the current inode. For the ".." dentry, we simply store it in the inode to avoid other unnecessary dependencies.

B-tree nodes are metadata structures to organize data blocks which contain file data. When a file is enlarged, *the newly written data and metadata structure modifications need to be persisted before the new file size is persisted.* These metadata structure modifications include B-tree height increases, B-tree node allocations and mod-

[3] It is not shown in the figure that if the dentry to remove is the head of the list, the pointer in the corresponding bucket is modified.

ifications, and data block allocations. Adding new data blocks and new inner B-tree nodes to the B-tree cannot be done atomically without copy-on-write. But even if this is not done atomically, it will not cause any problems since all the changes are not visible to users before the update of the file size. The B-tree root and height are stored in the inode and can be persisted atomically.

When a file is truncated, *the reduced file size shall be firstly persisted before reduced file data space is reclaimed.* For efficiency, the reclamations of these space are usually delayed for later file size increases unless the free space in the file systems is nearly exhausted.

3.4 Pointer-based Dual Views

One key property of soft updates is that it ensures that metadata present in memory always reflects the latest state of the file systems (i.e., the latest view) and metadata persisted in disks is consistent (i.e., the consistent view). The *dual views* technique allows delayed persistency and eliminates file system checking after a crash. Soft updates implements *dual views* based on page cache. However, as memory becomes storage, the page cache is removed for performance concerns in most NVMFS.

Thus, to provide *dual views* in NVMFS, a naive approach is to bring back the "cache" for metadata by maintaining another copy of the file system metadata. This approach, however, doubles the memory usage and causes unnecessary memory copies when synchronizing metadata between the latest view and the consistent view. For example, the whole persisted metadata structure has to be copied to its cache before the modification.

To implement efficient *dual views*, we propose pointer-based *dual views* in which most structures are shared by both views and different views are observed by following different pointers. We will then describe how different metadata structures (shown in Table 1) are designed to provide pointer-based *dual views* in SoupFS.

Inodes are already duplicated since VFS has its own data structure (VFS inode) to present an inode. So SoupFS uses its own inodes to store the consistent view and the corresponding VFS inode for the latest metadata.

Filenames are immutable in SoupFS. A filename always co-exists with its dentry and this binding relation never changes before the removal of the dentry. Thus the filename of a dentry can be directly shared by both views.

Dentries are almost read-only. During insertion, a dentry is inserted to the head of the linked list in the corresponding bucket. This procedure modifies no existing dentries. When removing a dentry, its predecessor is modified to point to the next dentry. SoupFS should be aware of such a modification so that it can traverse the list without the removed dentry in the latest view. At the same time, the removed dentry should be still observable in the consistent view if the whole removal has not been persisted. Otherwise, a crash might leave the file system inconsistent when there are multiple not-yet-persisted insertions and removals.

To share dentries in both views, SoupFS stores a pair of next pointers, *latest next* and *consistent next*, in a dentry. With these two next pointers, a traversal in the latest view is done by following the *latest next* pointer if it is non-null. Otherwise, the *consistent next* pointer is used. This guarantees that the latest information is retrieved by following the latest-next-first rule and the consistent view is observed by following only the *consistent next* pointers. Since the *latest next* is also stored in NVM, to differentiate the real *latest next* and the leftover *latest next* of a crash, SoupFS embeds an epoch number in the *latest next*. The epoch number is increased after a crash and *latest next* pointers with old epoch numbers are treated as null. This on-demand checking prevents after-crash stop-the-world checking in which all leftover *latest next* pointers are nullified.

Directory hash table buckets are changed upon an insertion to the dentry list or the removal of the last dentry. To provide two views, we maintain a *latest bucket* for each of the buckets and if not null, it always points to the latest first dentry in the dentry list. A *latest bucket* and its corresponding real bucket together act similarly to the two next pointers in dentries. For convenient memory management, all *latest buckets* for a hash table are gathered together in a volatile page and allocated on demand.

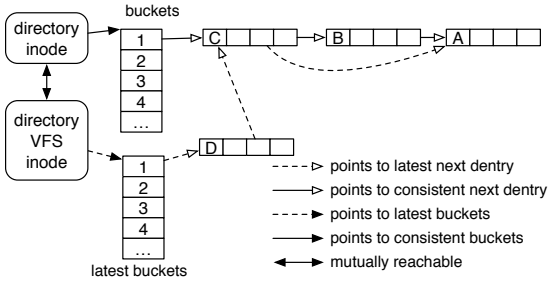


Figure 7: *Dual views* in a directory

An example is shown in Figure 7, in which the latest view can be observed by following dashed arrows and the consistent view is organized by solid arrows. We can also know from Figure 7 that dentry D is recently inserted and B is removed from the directory and both the insertion of D and removal of B have not been persisted yet[4].

B-tree nodes and data blocks: SoupFS focuses on metadata and does not protect consistency of written data, which is the same as soft updates[5]. SoupFS does not provide two views for B-tree nodes and data blocks. Nevertheless, there are still two views of file data in SoupFS.

[4] Since modifications to D and C are directly written in NVM, these changes might have been persisted. But these changes will be ignored after a crash since they are not observable in the consistent view.

[5] Even though soft updates can leverage page cache to provide two copies of data, it cannot guarantee a write spanning two blocks can be persisted atomically.

Table 2: Operations tracked by SoupFS

OP Type	Recorded Data Structures
diradd	added dentry, source directory*, overwritten inode*
dirrem	removed dentry, destination directory*
sizechg	the old and new file size
attrchg	-

One is the latest file data that are available by inspecting the B-tree root and the file size stored in VFS inode. The other is the persisted file data which can be obtained by following the B-tree root and the file size in SoupFS's inode. These two B-tree roots and file sizes form two B-trees that are built on the same set of B-tree nodes and data blocks. However, neither data in two B-trees are guaranteed to be consistent after a crash. To provide data consistency in SoupFS, techniques like copy-on-write can be adopted.

Allocator: The allocation information in NVM bitmap presents the consistent view and in-DRAM free-lists provide the latest view.

3.5 Dependency Tracking

Dependency tracking is one of the key parts of soft updates and is much simplified in SoupFS. Thanks to the byte-addressability of NVM, there are no more cyclic dependencies in the system. We thus can use a DAG to present dependencies among different parts of the file system, according to the paper of soft updates [10]. However, since SoupFS abandons the page cache and the block layer, the gap between the page cache and the file system disappear. In other words, a *persister* can know which operation on which structure needs to be persisted. By endowing with file system semantics, dependency tracking and enforcement are further simplified.

Dependency Tracking Structures: Although soft updates tracks dependencies in byte-level granularity, it is still a block-oriented dependency tracking that uses additional structures to track the dependencies among different blocks. Different from the original soft updates, SoupFS uses an inode-centric semantic-aware way to organize all dependencies.

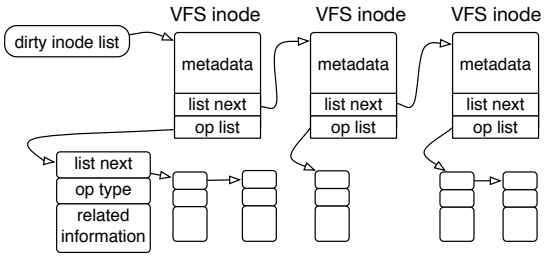


Figure 8: Dependency tracking structures

Figure 8 shows dependency tracking structures in SoupFS. In each VFS inode, an operation list (op list) tracks all recent operations on this inode that are not yet persisted. Each of the operations consists of an operation type (op type) and related information such as pointers to data structures involved during the operation. Table 2 shows the operations SoupFS tracks in detail, in which *diradd* and *dirrem* are used to track in-directory operations, *sizechg* is for regular file structure changes and *attrchg* is for attribute-only updates of an inode. An operation is created and inserted to the operation list of the VFS inode during the syscalls (see §3.6). Once the operation list is not empty, the VFS inode is added to the dirty inode list, waiting to be handled by the persisters.

Tracking these operations is sufficient for dependency enforcement. Supposing a VFS directory inode contains a *diradd*, by checking *added dentry*, SoupFS knows almost everything about the operation, e.g., the filename and the new inode. SoupFS can then persist these operations in the correct order that enforces update dependencies.

3.6 POSIX Syscalls

SoupFS classifies POSIX syscalls into the following categories and handles them accordingly.

Attribute-only Modification: These syscalls, *chmod* and *chown* for instance, only change the attributes of an inode. SoupFS handles these syscalls by directly updating the attributes in the corresponding VFS inode and insert an *attrchg* into the inode's operation list.

Directory Modification: Syscalls like *create*, *mkdir*, *unlink* and *rmdir* modify the content of a parent directory. SoupFS handles these syscalls according to steps described in §3.3. Then it inserts a *diradd* or a *dirrem* to the directory inode's operation list. The affected dentry is recorded as related data in the operation.

File Data Modification: These syscalls might modify the B-tree structures of a regular file. Examples include *write* and *truncate*. The deallocations of nodes are delayed, and the allocations and attachments of B-tree nodes and data blocks are directly done in the B-tree. The new file size and new file root (if changed) are updated only in the latest view (VFS inode). Finally, a *sizechg* is inserted into the inode's operation list.

Rename: *Rename* is special since it can involve more than one directory. Same as soft updates, SoupFS treats *rename* as a combination of a creation in the destination directory and a removal in the source directory. As a result, two operations, *diradd* and *dirrem* are inserted into the operation lists of the destination directory inode and the source directory inode respectively. Soft updates's ordering requirement for *rename* is also adopted by SoupFS, i.e., the persistence of the creation should be completed before the persistence of the removal. To track this dependency, *source directory* and *destination directory* are respectively recorded in *diradd* and *dirrem* as shown in Table 2.

If an existing dentry is overwritten in *rename*, it is directly modified by replacing its inode pointer. To reclaim

* These structures are only for rename.

the original inode in the overwritten dentry, SoupFS records it in *diradd* as the *overwritten inode*.

3.7 Dependency Enforcement

Dependency enforcement is done by daemons called *persisters*. *Persisters* periodically wake up, retrieve all dirty inodes from the dirty inode list, and persist each operation in the operation list according to the ordering requirements. The wake-up frequency can be specified at mount time as the bound of persistence.

It is simple to persist an operation. For *diradd*, a *persister* first ensures the persistence of the allocations and new structures. Then, it reflects the operation in the consistent view by updating the corresponding *consistent pointer* with the *latest pointer*.

For *dirrem*, a *persister* first makes the target data structure persistently removed from the consistent view, then reclaims memory used by the removed data structures.

For *sizechg*, a *persister* can get the newly allocated B-tree nodes and data blocks by inspecting the old and the new file size. Allocations are firstly persisted and then data blocks and B-tree nodes are persisted in a correct order. The file size, the B-tree root and height in the consistent view are updated only after all modifications within the B-tree are persisted. If it is a truncation, the truncated B-tree nodes and data blocks can be reclaimed after the persistence of the new file size, B-tree root and height. As an optimization, the reclamation is delayed for later file size increases.

For *attrchg*, attributes in the VFS inode are persistently written to the consistent inode. The atomicity of these operations is discussed in §3.8.

Finally, the persisted operation is removed from the operation list and the VFS inode is removed from the dirty inode list when its operation list is empty.

In the implementation, we deploy one *persister* for each NUMA node to prevent expensive cross-NUMA memory accesses.

3.8 Atomicity

SoupFS assumes that the failure-atomic write unit of NVM is a cache line, which is no less than 64 bytes. Based on this assumption, there are two kinds of atomic writes used in SoupFS.

The most commonly used one is an atomic update of a pointer, which is done using atomic operations provided by CPU. The other write is persisting an inode. Since the inode size in SoupFS is 64 bytes which is the cache line size, SoupFS needs to prevent the cache line from being evicted before updates to the inode are finished. This is guaranteed by using Intel RTM technology which will hold the cache line in cache until the transaction ends. Per-CPU cache-line-sized journals can be used as fallbacks of RTM to guarantee progress.

Both kinds of atomic writes only guarantee that an update is not partially persisted. It is the *clflush/clflushopt+sfence* that can guarantee the persistence of the update.

3.9 File System Checking

SoupFS is the same as soft updates in file system checking and recovery. Thanks to the consistent view, SoupFS can be instantly used after a crash without having to wait for file system checking or recovery. But in order to collect the memory leaked by the crash, a specialized *fsck* needs to be invoked manually. The *fsck* traverses the whole file system from the root and reconstructs the allocation bitmaps in which all allocated memory is useful.

3.10 Endurance

Although write endurance is not the design goal of SoupFS, we expect better write endurance than other NVMFS, since SoupFS eliminates journaling mechanisms which frequently need to write temporary backup data in NVM. In SoupFS, almost all writes to NVM are to modify the persistent state of the file system. The only exception is updates to the *latest next* pointer in dentries. While storing *latest next* in DRAM can further benefit the endurance, it involves additional data structures to track the relation between dentries and *latest next* pointers, which is complex. Moreover, the negative effect of updating *latest next* pointers is limited since in the implementation the update only happens in removal operations and it is likely to be kept in the cache before the operation is persisted by the persisters.

4. Evaluation

4.1 Experimental Setup

To evaluate the performance of SoupFS, we run micro-benchmarks and macro-benchmarks with a dual-socket Intel Xeon E5 server equipped with NVDIMM. Each 8-core processor runs at 2.3 GHz with four DDR4 channels. There are 48 GB DRAM and 64 GB NVDIMM equipped on the server, and we use one pmem device whose 32GB NVDIMM locate on one NUMA node in the evaluation.

We compare SoupFS against four Linux file systems: EXT4, EXT4 with DAX (EXT4-DAX), PMFS and NOVA. EXT4 can be directly used in Linux 4.9.6, but PMFS and NOVA need simple modifications to run on Linux 4.9.6.

PMFS, NOVA, and SoupFS obtain a range of NVM from kernel driver and manage NVM independently. EXT4-DAX bypasses the page cache using the DAX interface exposed by the persistent memory driver. We evaluate EXT4 only for reference since it cannot guarantee crash consistency in NVM. We provide no comparison with the original soft updates as there is no available soft updates implementation in Linux and simply running

FreeBSD UFS with soft updates on NVM cannot guarantee consistency due to the lack of block write atomicity.

Table 3: Micro-benchmark characteristics

Name	Workload
filetest	(I) create ($10^4 \times 100$) (II) unlink ($10^4 \times 100$)
dirtest	(I) mkdir ($10^4 \times 100$) (II) rmdir ($10^4 \times 100$)

4.2 Micro-benchmarks

We use two single-threaded micro-benchmarks to evaluate the throughput and latency of SoupFS, as shown in Table 3. The benchmarks run 100 iterations and in each iteration, the *filetest* creates 10^4 files in one directory and then deletes all of them. The *dirtest* is similar to the *filetest* but it creates directories instead of files.

Figure 9(a) and 9(b) show the throughput and latency of *create*, *unlink*, *mkdir* and *rmdir* tested using the *filetest* and *dirtest*. Generally, SoupFS performs best in all these tests. It outperforms NOVA in throughput by 43% to 405%, and reduces latency by 30% to 80%. We attribute the performance improvement to the reduction of flushes in the system call path. NOVA also performs well in the tests since it leverages in-DRAM radix trees to manage its directories. However, it still needs logs and cache flush operations to guarantee crash consistency, which causes relatively worse performance than SoupFS. Besides journaling and excessive flush operations, PMFS has high latency and low throughput also because its cost of directory lookup grows linearly with the increasing number of directory entries. This is notable for *create* and *mkdir* since one insertion to the directory needs to scan all existing dentries to find an available slot. For *unlink* and *rmdir*, the latency is very low since our benchmarks delete the files/directories in the same order they are created. If the dentry preceding the to-be-removed dentry is not in use, PMFS will merge those two dentries during the removal. Thus in *unlink* and *rmdir*, PMFS needs to search at most two dentries to find the dentry to remove, yielding low latencies as shown in the figure. EXT4 and EXT4-DAX leverage hashed B-trees to speed up directory accesses, thus they achieve better performance than PMFS.

Figure 9(c) shows the latency distribution for *create* in the *filetest*. Latencies longer than 30us are not shown in the figure for clarity. The result proves the average latencies shown in Figure 9(b). Most of the latencies for SoupFS locate at around 3us and latencies for NOVA at around 4us. Due to the inefficient directory organization, latencies for PMFS evenly distribute and steadily rise as the number of files in a directory increases.

Table 4: Filebench workload characteristics

Workload	Average file size	# of files	I/O size	r:w ratio
Fileserver	128KB	10000	1M	1:2
Fileserver-1K	1KB	10000	1M	1:2
Webproxy	16KB	10000	16K	5:1
Varmail	16KB	5000	1M	1:1

4.3 Macro-benchmarks

We evaluate the performance of SoupFS for real world applications by running a set of macro-benchmark workloads, including Filebench and Postmark.

Filebench: Filebench is a file system benchmark that simulates a large variety of workloads by specifying different models. We integrate the recent fixes to Filebench by Dong et al. [7] to make our evaluation more accurate. Table 4 shows the characteristics of Filebench workloads. We run these benchmarks from 1 to 20 threads multiple times and report the average to show the throughput and scalability. The coefficient of variation is 1.8% in average.

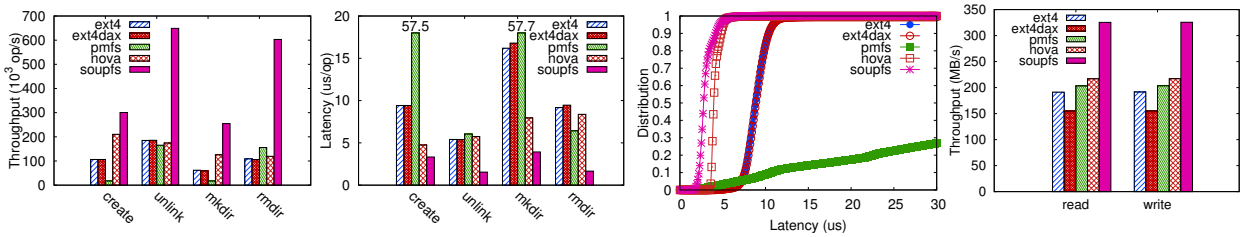
As shown in Figure 11(a) to 11(d) SoupFS performs best in general. The performance drop after eight threads is caused by the NUMA architecture. When the number of threads exceeds eight, either the threads contend on eight cores of a NUMA node, or there are a lot of cross-NUMA memory accesses. We thus evaluate with Filebench bound to one NUMA node and report the result in Figure 12(a) to 12(d), in which the throughput still cannot scale well, but the performance drop disappears.

The throughput of fileserver is lower than those of other Filebench workloads. This is because the default average file size is 128KB, causing each operation to write more data and the data write speed dominates. SoupFS performs slightly better in this workload since it provides two views of the file size so that it does not need to persist the B-tree structures immediately. As a drawback, the file data are not guaranteed to be persisted after the *write* syscall, which is different from other NVMFS. We also evaluate fileserver with 1K file size to highlight the metadata operations (Figure 11(b)). The throughput of all file systems increases and SoupFS outperforms NOVA by up to 89% and PMFS by up to 47%.

The webproxy involves recreating and reading several files in a directory with many files. PMFS performs worst due to its inefficient directory access, while other file systems, by using hash tables (SoupFS), radix trees (NOVA) and hashed B-trees (EXT4 and EXT4-DAX), perform much better. SoupFS performs slightly better when there are fewer threads because of metadata operations like file removals and creations.

The varmail acts as a mail server on which users read, remove, reply, and write mails. In this workload, *fsync* operations eliminate the benefit of page cache in EXT4 and the performance of PMFS is limited by its slow directory design. SoupFS outperforms NOVA by up to 75% due to fast metadata operations.

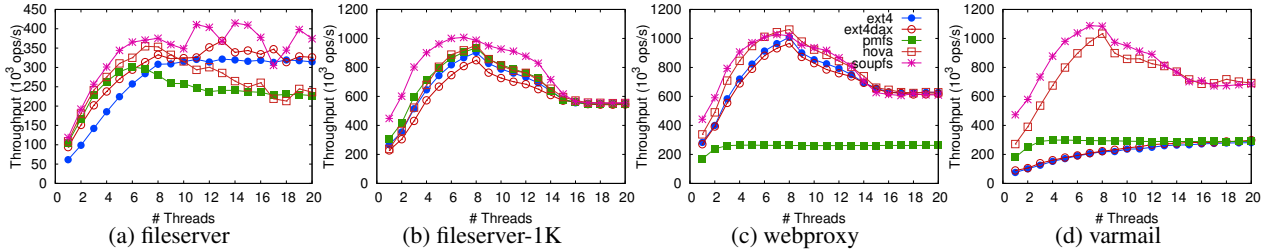
Postmark: Postmark is a benchmark to simulate mail servers. We enlarge the number of transactions to 10^6 in the default single-threaded Postmark configuration to test the performance. Figure 10 shows that SoupFS outperforms other file systems by about 50%.



(a) Throughput (b) Latency (c) Latency CDF for *create*

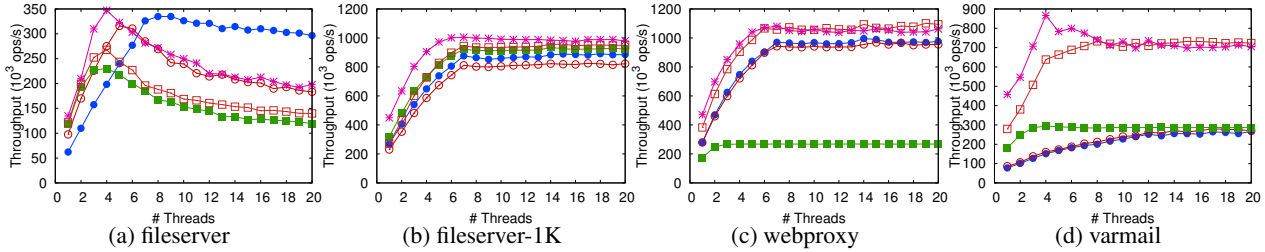
Figure 9: Throughput and latency of file system operations

Figure 10: Postmark



(a) fileserver (b) fileserver-1K (c) webproxy (d) varmail

Figure 11: Throughput of Filebench (EXT4 does not guarantee correctness)



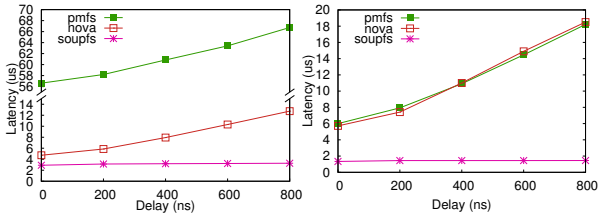
(a) fileserver (b) fileserver-1K (c) webproxy (d) varmail

Figure 12: Throughput of Filebench with NUMA binding (EXT4 does not guarantee correctness)

4.4 Sensitivity to NVM Characteristics

Different NVM technologies have different write latencies. The NVDIMM we use has the same performance as DRAM; however, NVM built by PCM and 3D XPoint is expected to have higher latency especially higher write latency than DRAM. We roughly evaluate the sensitivity to NVM write latency of different file systems by inserting delays after each *clflush* instruction.

Figure 13 shows the latency of *create* and *unlink* in the *filetest* micro-benchmark with different delays inserted after *clflush*. In both cases, the latency of SoupFS remains unchanged with increasing delays due to the elimination of cache flushes in the critical path. The latency of PMFS and NOVA increases because they need cache flushes for crash consistency during the syscall. Increasing the delay from 0 to 800ns, the latency of NOVA increases 8us which is nearly 200% of its original value for *create* (Figure 13(a)). The increased value matches our estimation in §2.2. Although the increased latency for PMFS is similar, the *create* latency of PMFS is still dominated by the slow directory lookup performance, so the relative influence is not significant. For *unlink* in Figure 13(b), both NOVA and PMFS are affected by the clflush delays, with latency increased from 6us to 18us.



(a) create (b) unlink

Figure 13: Latency of *filetest* with different clflush delays

clflushopt, the asynchronous and parallel version of *clflush*, is not available on our evaluation platform. We thus give no evaluation of *clflushopt*. However, since most of the cache flushes in the file systems are for persistency ordering guarantee, file systems usually use a combination of *clflushopt+sfence* which neutralizes the merit of asynchronism and parallelism. Other approaches, like running instructions between *clflushopt* and *sfence* or using existing atomic operations to replace fences, are feasible but they can only be used under certain conditions where appropriate instructions and existing atomic operations are available. *clwb*, which is not available on our platform either, is similar to *clflushopt* but keeps the data in the cache after persisting it. However, the merit of *clwb* can barely bring performance improvement since existing NVMFS are designed to avoid re-accessing flushed cache lines in one syscall.

We would also like to compare the impact of different NVM bandwidth on the performance of SoupFS. Unfortunately, we fail to change the BIOS configuration of the PCIE extended area as suggested by Sengupta et al. [37], as it is inaccessible to a normal user. We thus leave such a comparison as future work. Yet, since the delayed persistency of SoupFS has fewer requirements for immediately persisting the data which urgently need bandwidth, we envision that SoupFS would gain more benefits compared to other alternatives like PMFS and NOVA that require synchronous flushes.

5. Related Work

Metadata update approaches: Other than soft updates, there have been various approaches to preserving metadata consistency, including shadow paging [14, 18, 30], log-structuring [17, 19, 24, 31, 33, 34], journaling [2, 12, 39] and WriteAtomic [29]. There have been various other ways to represent write ordering, using backpointers [3], transactional checksums [28], patches [9]. For example, NoFS [3] proposes backpointers to reduce ordering requirement when persisting data. It, however, requires adding a backpointer to not only metadata but also data, which increases storage overhead. Moreover, a key assumption is that a backpointer and its data or metadata are persisted atomically, a property not available in NVM.

NVM-aware file systems: Some designs have considered using NVM to accelerate metadata update performance. For example, Network appliance's WAFL [14] leverages NVRAM to keep logs to improve synchronous log update performance. The Rio cache [27] enabled by uninterruptible power supply can also be used to log synchronous metadata updates with high performance. However, even with NVM as cache, they may still suffer from consistency issues from unanticipated cache line eviction for metadata updates.

The promising feature of NVM has stimulated the design and implementation of several recent NVM file systems such as BPFS [4], SCMFS [44], Aerie [41], EXT4-DAX [5, 6], NOVA [45] and HiNFS [25]. Generally, they allow "execute in place" (XIP) to bypass the block layer and page cache to reduce management overhead, or provide a buffer cache with in-place commit feature [20].

Wu and Zwaenepoel describe eNVy [43], a storage system that directly presents flash memory as a linear address space into memory bus using paging translation. To overcome slow write performance of flash memory, eVNy uses a small battery-backed SRAM as a buffer to create a copy of the updated page to give the illusion of in-place update. As the NVM nowadays could achieve within one order of magnitude speed of DRAM, SCMFS [44] and SIMFS [38] further directly map a file data space into the virtual address space of a process. These techniques are orthogonal to the design of SoupFS.

Data structures for NVM: Venkataraman et al. [40] describe a persistent B+ tree implementation for NVM, yet requires synchronous flushes at each update path. NV-Tree [46] instead uses DRAM as indexes to reduce synchronous flushes cost, but requires scanning all NVM in order to reconstruct the indexes upon a crash. Mnemosyne [42] provides a transactional interface for consistent updates of application data structures. SoupFS eliminates cache flushes in the critical path of file system operations and need no journaling for crash consistency.

Crash consistency and memory persistency models: Chidambaram et al. [2] propose separating ordering from durability and introduce optimistic crash consistency by leveraging a hypothetical hardware mechanism called asynchronous durability notification (ADN). SoupFS can be made simple and efficient with ADN by avoiding flushing already persistent cache lines.

Foedus [15] leverages the duality of volatile pages and stratified snapshot pages to provide snapshots and crash consistency in an NVM-based in-memory database. Most of the pointers in Foedus are dual-page-pointers stored together. SoupFS uses a similar technique like "dual pointers" to present *dual views* of the file system metadata in some structures like dentries. However, the latest pointers for the latest view may be created on demand and stored separately from the consistent pointer in SoupFS.

Pelley et al. [26] introduce the concept of memory persistency as an analogy of memory consistency, summarize a set of persistency models such as strict and epoch persistency and additionally introduce strand persistency. Kolli et al. [16] further describe a set of techniques like deferring commit until lock release to different persistency models to relax write orderings for transactions whose read/write sets are known in advance. Unlike prior work, SoupFS extends soft updates instead of logging for ensuring the persistency models.

6. Conclusions

This paper describes SoupFS, a soft updates implementation for NVM. SoupFS is made simple by leveraging byte-addressability to simplify dependency tracking and enforcement. SoupFS is made fast through delaying most synchronous flushes from the critical path thanks to the efficient pointer-based *dual views*. Evaluations show that SoupFS outperforms state-of-the-art NVMFS.

Acknowledgment

We thank our shepherd Liuba Shrira and the anonymous reviewers for their constructive comments. This work is supported in part by National Key Research & Development Program of China (No. 2016YFB1000104), China National Natural Science Foundation (No. 61572314, 61525204 and 61672345), Zhangjiang Hi-Tech program (No. 201501-YP-B108-012) and CREATE E2S2.

References

[1] V. Aurora. Soft update, hard problems. https://lwn.net/Articles/339337/, 2009.

[2] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 228–243. ACM, 2013.

[3] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency without ordering. In *FAST*, page 9, 2012.

[4] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146. ACM, 2009.

[5] Supporting filesystems in persistent memory. https://lwn.net/Articles/610174/, 2014.

[6] Support ext4 on nv-dimms. http://lwn.net/Articles/588218/, 2014.

[7] M. Dong, Q. Yu, X. Zhou, Y. Hong, H. Chen, and B. Zang. Rethinking benchmarking for nvm-based file systems. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 20:1–20:7, 2016.

[8] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, 2014.

[9] C. Frost, M. Mammarella, E. Kohler, A. de los Reyes, S. Hovsepian, A. Matsuoka, and L. Zhang. Generalized file system dependencies. In *SOSP*, pages 307–320. ACM, 2007.

[10] G. R. Ganger, M. K. McKusick, C. A. Soules, and Y. N. Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems (TOCS)*, 18(2):127–153, 2000.

[11] G. R. Ganger and Y. N. Patt. Metadata update performance in file systems. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*. USENIX Association, 1994.

[12] R. Hagmann. Reimplementing the cedar file system using logging and group commit. In *SOSP*. ACM, 1987.

[13] V. Henson. Khb: A filesystems reading list. https://lwn.net/Articles/196292/, 2006.

[14] D. Hitz, J. Lau, and M. A. Malcolm. File system design for an nfs file server appliance. In *USENIX winter*, volume 94, 1994.

[15] H. Kimura. Foedus: Oltp engine for a thousand cores and nvram. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 691–706. ACM, 2015.

[16] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 399–411. ACM, 2016.

[17] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai. The linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review*, 40(3):102–107, 2006.

[18] E. Kustarz. Zfs-the last word in file systems. *http://www.opensolaris.org/os/community/zfs/*, 2008.

[19] C. Lee, D. Sim, J. Hwang, and S. Cho. F2fs: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, 2015.

[20] E. Lee, H. Bahn, and S. H. Noh. Unioning of the buffer cache and journaling layers with non-volatile memory. In *USENIX Conference on File and Storage Technologies*, pages 73–80, 2013.

[21] R. Liu and H. Chen. Ssmalloc: a low-latency, locality-conscious memory allocator with stable performance scalability. In *Proceedings of the Asia-Pacific Workshop on Systems*, page 15. ACM, 2012.

[22] K. McKusick. Journaling soft updates. In *BSDCan*, 2010.

[23] M. K. McKusick, G. R. Ganger, et al. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *USENIX Annual Technical Conference, FREENIX Track*, pages 1–17, 1999.

[24] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. Sfs: random write considered harmful in solid state drives. In *FAST*, page 12, 2012.

[25] J. Ou, J. Shu, and Y. Lu. A high performance file system for non-volatile main memory. In *European Conference on Computer Systems*, 2016.

[26] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *ISCA*, pages 265–276. ACM, 2014.

[27] C. PM, N. WT, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The rio file cache: Surviving operating system crashes. In *ASPLOS*, 1996.

[28] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Iron file systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 206–220. ACM, 2005.

[29] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou. Transactional flash. In *OSDI*, pages 147–160, 2008.

[30] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9, 2013.

[31] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[32] A. M. Rudoff. Deprecating the pcommit instruction. https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction, 2016.

[33] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured memory for dram-based storage. In *Proceed-*

ings of the 12th USENIX Conference on File and Storage Technologies (FAST 14), pages 1–16, 2014.

[34] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for unix. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, pages 3–3. USENIX Association, 1993.

[35] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. Soules, and C. A. Stein. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *USENIX Annual Technical Conference, General Track*, pages 71–84, 2000.

[36] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *Usenix ATC*, 2000.

[37] D. Sengupta, Q. Wang, H. Volos, L. Cherkasova, J. Li, G. Magalhaes, and K. Schwan. A framework for emulating non-volatile memory systemswith different performance characteristics. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pages 317–320. ACM, 2015.

[38] E. Sha, X. Chen, Q. Zhuge, L. Shi, and W. Jiang. Designing an efficient persistent in-memory file system. In *IEEE Non-Volatile Memory System and Applications Symposium*, pages 1–6, 2015.

[39] Silicon Graphics International Corp. Xfs: A high-performance journaling file system. http://oss.sgi.com/projects/xfs, 2012.

[40] S. Venkataraman, N. Tolia, P. Ranganathan, R. H. Campbell, et al. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, volume 11, pages 61–75, 2011.

[41] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, , and M. M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth Euro- pean Conference on Computer Systems (EuroSys 14)*, page 14:114:14, 2014.

[42] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. *ACM SIGPLAN Notices*, 46(3):91–104, 2011.

[43] M. Wu and W. Zwaenepoel. envy: A non-volatile, main memory storage system. In *ASPLOS*, 1994.

[44] X. Wu and A. Reddy. Scmfs: a file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 39. ACM, 2011.

[45] J. Xu and S. Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *USENIX Conference on File and Storage Technologies*, pages 323–338, 2016.

[46] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, 2015.

[47] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 421–432. ACM, 2013.

SmartMD: A High Performance Deduplication Engine with Mixed Pages

Fan Guo[1], Yongkun Li[1,2], Yinlong Xu[1,3], Song Jiang[4], John C. S. Lui[5]
[1]*University of Science and Technology of China*
[2]*Collaborative Innovation Center of High Performance Computing, NUDT*
[3]*Anhui Province Key Laboratory of High Performance Computing, USTC*
[4]*University of Texas, Arlington*
[5]*The Chinese University of Hong Kong*

Abstract

In hypervisor-based virtualization environments, translation lookaside buffers (TLBs) misses may induce two-dimensional page table walks, which may incur a long access latency, and this issue becomes worse with ever increasing memory capacity. To reduce the overhead of TLB misses, large pages (e.g., 2M-pages) are widely supported in modern hardware platforms to reduce the number of page table entries. However, memory management with large pages can be inefficient in deduplication, leading to low utilization of memory, which is a precious resource for a variety of applications.

To simultaneously enjoy benefits of high performance by accessing memory with large pages (e.g., 2M-pages) and high deduplication rate by managing memory with base pages (e.g., 4K-pages), we propose Smart Memory Deduplciation, or SmartMD in short, which is an adaptive and efficient management scheme for mixed-page memory. Specifically, we propose two lightweight schemes to accurately monitor pages' access frequency and repetition rate, and present a dynamic and adaptive conversion scheme to selectively split or reconstruct large pages. We implement a prototype system and conduct extensive experiments with various workloads. Experiment results show that SmartMD can simultaneously achieve high access performance similar to systems using large pages, and achieves a deduplication rate similar to that applying aggressive deduplication scheme (i.e., KSM) at the same time on base pages.

1 Introduction

In modern computers, processors use page tables to translate virtual addresses to physical addresses. To accelerate the translation, TLB was introduced to cache virtual-to-physical address mappings. While TLB is critical to system's performance, its misses carry high penalty for accessing the page table in memory. In particular, in a system employing hypervisor-based virtualization, the hypervisor and guests maintain separate page tables, and TLB misses will lead to high-latency two-dimensional page table walks. Previous works [13, 17] show that this is often the primary contributor to the performance difference between virtualized and bare-metal systems. In fact, the overhead of TLB misses has become one of the primary bottlenecks of memory access.

Moreover, while memory size becomes increasingly larger, TLB's capacity cannot grow at the same rate as DRAM. To reduce TLB miss ratio, large pages are introduced in many modern hardware platforms to reduce the number of page table entries required to cover a large memory space. For example, the X86 platform supports 2M-page and 1G-page, while the ARM platform supports 1M-page and 16M-page [9].

It is important to note that different VMs on the same host machine often run similar operating systems (OSes) or applications. It is highly likely that there exists a great deal of redundant data among different VMs [14]. Thus, we can save memory space by removing redundant data and sharing only a single copy of the data among different VMs (also known as memory deduplication). However, for memory systems with large pages (e.g., 2M-pages), our experiments show that it is hard to find duplicate large pages even the memory contains a large amount of redundant data. In other words, deduplication in unit of the large page is ineffective and usually saves only a small amount of memory space.

To enable more effective deduplication, current OSes exploit an aggressive deduplication approach (ADA), which aggressively splits large pages (e.g., 2M-pages) to base pages (e.g., 4K-pages) and performs deduplication among base pages [22]. However, after the splitting, the memory space covered by translation entries in the TLB can be significantly reduced. Although ADA saves more memory space, accessing the split large pages significantly increases TLB miss ratio and degrades access performance. Moreover, the reconstruction of split large

pages is not well supported in current OSes. In a system that keeps running, there are increasingly more split pages, leading to continuous degradation of memory access performance.

In this paper, our objective is to maximize memory saving with deduplication while keeping high memory access performance on a server hosting multiple VMs. In particular, we propose SmartMD, to maximize memory saving while keeping high performance of memory access. The main idea is to split cold large pages with high repetition rate to save memory space, and at the same time, to reconstruct split large pages when they become hot to improve memory access performance. The key challenges are how to efficiently monitor repetition rate and access frequency of pages, and how to dynamically conduct conversions between large pages and base pages so as to achieve both high deduplication rate and high memory access performance. The main contributions of this work can be summarized as follows.

- We propose two lightweight schemes to monitor pages on their access frequency and repetition rate. In particular, we introduce counting bloom filters and sampling into the monitor such that it can accurately track pages' status with very low overhead. Additionally, we propose a labeling method to identify duplicate pages during the monitoring, which can greatly accelerate the deduplication process.
- We propose an adaptive conversion scheme which selectively splits large pages to base pages, and also selectively reconstructs split large pages according to the access frequency and repetition rate of these pages and memory utilization. With this bidirectional conversion, we can take both benefits of high memory access performance with large pages and high deduplication rate with base pages.
- We implement a reconstruction facility by selectively gathering scattered subpages of a split large page, and then carefully re-create descriptor and page table entry of the split large page. As a result, the memory access performance can be improved by reconstructing split large pages which turn hot.
- We implement a prototype and conduct extensive experiments to show the efficiency of SmartMD. Results show that SmartMD can simultaneously achieve high memory access performance similar to that of large page-based systems, and high deduplication rate similar to that produced by aggressive deduplication schemes, such as KSM.

The rest of the paper is organized as follows. We introduce the background of memory virtualization, large page management, and current aggressive deduplication technology in Section 2. We motivate the design for improving the aggressive deduplication policy in Section 3. In Section 4, we discuss the design of various techniques used in SmartMD. In Section 5, we describe the experiment setup and present the evaluation results to show the efficiency of SmartMD. Section 6 discusses the related work and Section 7 concludes the paper.

2 Background

2.1 Memory Virtualization

To efficiently utilize limited memory space, a high-performance server hosting virtual machines (VMs) usually dynamically allocates its memory pages to VMs on demand. Because of the dynamic allocation, physical addresses of the memory pages allocated to a VM are often not contiguous. So in a hypervisor-based virtualized system, a VM uses guest's virtual addresses (GVA) and guest's physical addresses (GPA) for its memory access. GPA are logical addresses on the host and they are mapped to host physical addresses (HPA). To improve the address translation performance from GPA to HPA, extended page tables (named by Intel) or nested page tables (named by AMD) [12] have been introduced. With the extended page tables [1], a VM will carry out a two-dimensional page walk to access its data with two steps. First, a GVA is translated to its corresponding GPA using guest's page tables. Second, the GPA is further translated to its corresponding HPA using extended page tables.

When using base pages (i.e., 4KB pages in X86-64 system), both the guest's page table and extended page table are composed of four levels. Accessing each level of the guest's page table will trigger the traversal of the extended page table. In the worst case, a two-dimensional page walk will require 24 memory references [12, 23], which is apparently unacceptable. A commonly practice to accelerate the address translation is to cache frequently used global mapping from GVA to HPA in the TLB [27].

However, when the memory becomes increasingly large the page tables consistently grow, and as a result the hit ratio of TLB reduces. To increase the hit ratio of TLB and further speedup the address translation in a system with a large amount of memory, large pages have been widely adopted in today's systems.

2.2 Advantage of Using Large Pages

A large page is composed of a fixed number of contiguous base pages. For example, in a X86-64 system, OS

[1] In this paper we will use Intel's extended page tables as an example, though the design and conclusions derived from the evaluation results are also applicable to the system using nested page tables.

Benchmark	Host: Base Guest: Large	Host: Large Guest: Base	Host: Large Guest: Large
SPECjbb	1.06	1.12	1.30
Graph500	1.26	1.34	1.68
Liblinear	1.13	1.14	1.37
Sysbench	1.07	1.09	1.20
Biobench	1.02	1.18	1.37

Table 1: Benchmark performance with selective use of large pages. Details of the benchmark programs are described in Section 5. The performance is normalized against the one for running the benchmark on the system using base pages in both guest and host OSes.

uses one 2MB-page entry to cover a contiguous 2MB region of memory for its address translation, instead of using 512 4KB-page entries to cover it. In a virtual environment, large pages can be supported in both guest's page tables and extended page tables [12]. With large pages, the page table becomes significantly smaller, and much larger memory space can be covered by a TLB table of the same size. In this way, using large pages helps to increase TLB hit ratio for global mappings. In particular, it reduces maximum number of memory references in a 2D page walk after a TLB miss from 24 to 15 [12].

To show improvement of memory access performance with large pages, we run experiments with various benchmarks in a KVM virtual machine. Detailed configuration of the virtual machine is described in Section 5, and we present the experimental results in Table 1. We can see that the performance can be significantly improved for most of the benchmarks even if we use large pages only in guest's OS or in host's OS. In particular, if we use large pages in both OSes, the performance of Graph500 is improved by 68% over the baseline system in which only base page is used.

2.3 Impact of Using Large Pages on Memory Deduplication

Usually there is a great deal of duplicated data residing in the memory of a virtualized machine [14]. Deduplication among different VMs will lower the memory demand and keep memory from being overcommitted. However, even though there can be plenty of duplicate data in the memory, there can be very few duplicate large pages. While the deduplication removes duplicate data at the granularity of page, it may not be effective with the use of large pages. Our experiments show that ADA can save 13.7% - 47.9% of used memory for the selected benchmarks, but deduplication in unit of large page saves only 0.8% - 5.9% of used memory (see Table 2). That is, deduplication among different VMs in unit of large page can save very little memory. Thus, major OSes support an aggressive deduplication approach (ADA), which splits a large page into base pages and then applies deduplication on base pages [22], such as KSM in Linux.

Policy	Benchmark	Memory Saving	Performance
Large Page w/o ADA	Graph500	0.37 GB(3.4%)	1
	SPECjbb2005	0.40 GB(5.9%)	1
	Liblinear	0.32 GB(2.0%)	1
	Sysbench	0.09 GB(0.8%)	1
	Biobench	0.20 GB(1.4%)	1
Large Page with ADA	Graph500	5.18 GB(47.9%)	0.695
	Specjbb2005	1.83GB(26.9%)	0.922
	Liblinear	3.79 GB (23.7%)	0.846
	Sysbench	2.83 GB(18.0%)	0.867
	Biobench	1.88 GB(13.7%)	0.910

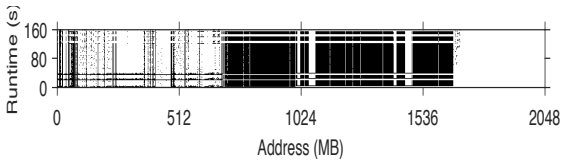
Table 2: Memory saving and performance of large-page-based systems with/without running ADA (aggressive deduplication approach), which splits all large pages. When ADA is not used, deduplication is applied at the large-page granularity. Memory saving is normalized against the memory demand in the system without using any deduplication. The performance is normalized against that for the system using large pages without applying ADA.

3 Motivations

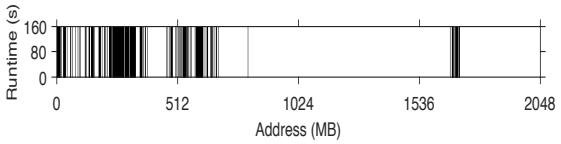
3.1 Aggressive Deduplication

When a large page contains duplicate subpages, ADA will split it into base pages and then perform deduplication on these base pages. Although ADA has the potential to save significant amount of memory space, the page tables become much larger, which will reduce the hit ratio of TLB and increase memory access time. The worst scenario of ADA is that it splits a large page that has high access frequency and low repetition rate (or the percentage of duplicate subpages belonging to the large page). And it compromises memory access performance and produces little memory saving.

We carry out experiments to show the statistics of memory pages. Fig. 1 shows the distributions of large pages with high access frequency or high repetition rate of a VM running SPECjbb. From Fig. 1(a), we can see that the SPECjbb benchmark constantly accesses some large pages throughout its entire run time while other large pages are rarely accessed. Fig. 1(b) shows that majority of large pages with high repetition rate appears only in few memory regions. Comparing Fig. 1(a) with Fig. 1(b), we find that many large pages have high access frequency but few duplicate subpages. In the meantime, there exist large pages with many duplicate subpages and low access frequency. In short, with ADA that selects large pages for splitting without considering page access frequency and repetition rate, the benefit of its limited memory saving can be more than offset by the degraded memory access performance for many applications.



(a) Location of large pages with high access frequency.



(b) Location of large pages with repetition rate higher than 1/8.

Figure 1: Memory usage of SPECjbb.

We experimentally compare memory saving and memory access performance in the system using large page and ADA. The results are shown in Table 2. With ADA, the system saves 13.7%-47.9% of memory space but is slowed down by up to 30.5% due to increased TLB misses after splitting large pages. Specifically, the percentage of large pages drops to 16% on average. On the other hand, retaining large pages preserves high memory access performance, but it loses opportunities of reducing memory usage. Thus, current memory management scheme is inadequate for virtualized systems running memory-intensive applications.

To this end, we propose SmartMD, a selective deduplication scheme that assigns each large page a priority of being split for potential deduplication according to its access frequency and repetition rate. SmartMD splits large pages with high repetition rate and low access frequency and performs deduplication among their subpages to save memory while maintaining high access performance.

3.2 Difficulties of Converting Base Pages to Large Pages

Major OSes support splitting of large pages to produce more deduplication opportunities. However, the reconstruction of base pages back into large pages is not well supported [22, 8]. In particular, only large pages whose base pages are not shared with those in other large pages can be reconstructed. Furthermore, the reconstruction may substantially compromise system performance. Meanwhile, instead of releasing free pages back to the host, a VM often keeps these pages for its incoming applications. Thus, the conversion of base pages to large pages in current OSes may cause incremental degradation of memory access performance. In this work, we propose an approach to efficiently reconstruct large pages to improve memory access performance.

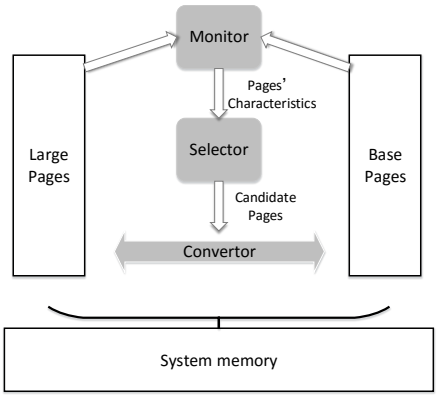


Figure 2: Illustration of SmartMD's Architecture.

3.3 The Challenges

Monitoring pages' statuses. SmartMD needs to track pages' access frequency and repetition rate, which are not directly disclosed by current OSes. Meanwhile, monitoring these parameters will introduce additional overheads. Thus, we need to design an efficient monitoring mechanism with low overhead.

Choosing right pages. Splitting large pages into base pages and reconstructing base pages into large pages may have big negative impacts on memory access performance. SmartMD must carefully select right pages to split and reconstruct for maximal efficacy and minimal side effect. Furthermore, applications' demands on memory and CPU may change dynamically, so SmartMD needs to identify current performance bottleneck and resource constraint and to provide an adaptive conversion mechanism between large pages and base pages to alleviate the situation.

Reconstructing large pages. SmartMD provides an approach to reconstruct base pages into large pages. However, implementation of the approach can be challenging, because splitting a large page not only changes its descriptor and page table entries of its subpages, but also breaks the contiguity of its subpages. Even worse, some subpages might have been freed after splitting, which imposes great difficulty on the reconstruction process.

4 Design of SmartMD

In this section we will overview the design of SmartMD followed with design details on each of its components.

4.1 Overview of SmartMD

As shown in Fig. 2, SmartMD is composed of three modules, *Monitor*, *Selector*, and *Converter*. In the Monitor, we provide two lightweight schemes to track number of

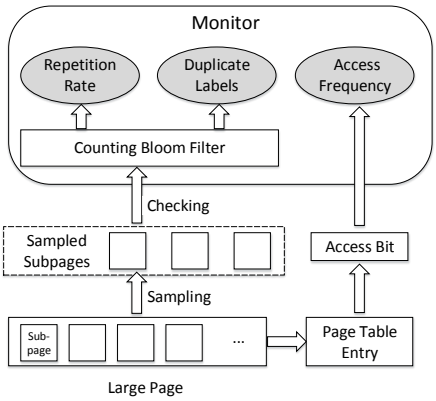


Figure 3: Design of the Monitor.

duplicate subpages, or the repetition rate for large pages. This information will be used by the Selector to select large pages for splitting or base pages for reconstruction. In particular, we propose an algorithm which dynamically performs the selection according to the current memory utilization, data access frequency, and large-page repetition rate. Finally, the Converter performs the conversion between large pages and base pages.

4.2 The Monitor

The Monitor uses a thread to periodically scan pages to measure memory utilization as well as page access frequency and repetition rate. Fig. 3 illustrates the techniques used in the Monitor and its workflow.

Monitoring memory utilization and page access frequency. We note that the OS already provides a utility to monitor and disclose the size of free memory space in a system. However, it does not provide a utility to directly monitor and disclose page access frequency. To address this issue, SmartMD periodically scans access bit of each page to gauge pages' access frequency. It clears the access bits of all pages at the beginning of a monitoring period, and checks each of them after *check_interval* seconds. If the access bit of a page is one, which is set due to a reference to the page in the period, SmartMD will increment its access frequency by one. Otherwise, the page was not accessed in the last period and its access frequency is decremented by one. If a large page has been split, we check the frequencies of its subpages and see if any of them is larger than zero. If yes, we increment frequency of the the original large page by one. However, we keep the frequency value always in the range from 0 to N, where N is a positive integer, and will not change it beyond the range. We initialize a page's frequency to $N/2$ when the system starts.

Detecting repetition rate of pages. To measure the repetition rate of a large page (or the percentage of duplicate base pages in the large page), existing approaches use comparison trees to identify duplicate pages [11]. However, they carry high CPU overhead. In contrast, SmartMD uses a counting bloom filter for an approximate identification.

The counting bloom filter is a one-dimensional vector, and each of its entries is a 3-bit counter. As shown in Fig. 4, when scanning a large page, SmartMD uses the counting bloom filter to check whether its subpages are duplicates or not. Specifically, when checking a subpage, SmartMD applies three hash functions on the subpage's content to calculate the indexes of its corresponding counters. For each subpage, SmartMD also records its signature, which is produced by applying a hash function on its content and is used to represent the page. If a page is checked for the first time (i.e., its recorded signature is not found), SmartMD will increase its corresponding counters by one. Otherwise, if all of the counters are greater than one, we consider this page as a duplicate one. If a page is modified, SmartMD decrements each of its current counters by one and increments each of its new counters by one. In addition, if a page is released, SmartMD also decrements each of its counters by one.

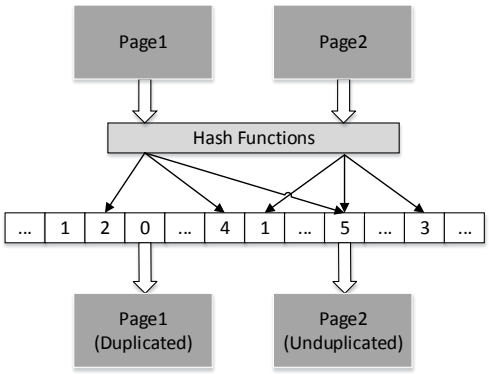


Figure 4: Identification of duplicate pages by using a counting bloom filter.

To make a trade-off between memory overhead and identification accuracy, SmartMD sets the size of the counting bloom filter, in terms of counters in it, as eight times of the number of base pages in the system. With this configuration, SmartMD can ensure that the false positive of the bloom filter is less than 3.06% [1].

SmartMD adopts a sampling-based approach to further accelerate the identification. Specifically, the Monitor first samples some subpages in a large page and calculates their hash values. It then checks whether these sampled subpages have been modified during the previous monitoring time period by comparing their current signatures with the ones on record. If a large page has been modified or is scanned for the first time during the sampling process, the Monitor will scan all the subpages

to update their signatures and insert them into the counting bloom filter. Meanwhile, SmartMD calculates the repetition rate of the large page. Otherwise, the Monitor calculates the repetition rate only among the sampled subpages, instead of all subpages in the large page, so as to reduce the overhead. For the subpages identified by the Monitor as duplicates, SmartMD labels them as a hint to the deduplication component to improve its efficiency. Specifically, when a large page is being split, SmartMD uses KSM to deduplicate redundant pages. KSM searches the labeled pages in the comparison trees to speed up the deduplication process. SmartMD organizes each large page's metadata about its access frequency and repetition rate in a linked list.

Our sampling-based detecting algorithm can help to substantially reduce the CPU overhead for most workloads. Experiments show that the ratio of misidentification of duplicate pages is less than 5% by sampling only 25% subpages in a large page. In particular, the counting bloom filter improves the efficiency of SmartMD on three aspects. First, it helps SmartMD to obtain approximate repetition rate of large pages with a small overhead. By using the repetition rate, we can avoid splitting large pages with low repetition rate. Second, it labels identified duplicate pages to speed up the deduplication process of SmartMD. Third, it reduces the number of nodes in the deduplicaiton trees by only splitting large pages with high repetition rate.

4.3 The Selector

To improve memory access performance, the Selector chooses candidate large pages for splitting based on two metrics, namely access frequency and repetition rate.

Identifying cold and hot pages. Upon knowing pages' access frequency from the Monitor module, the Selector divides all pages into three categories, cold, warm, and hot, with two thresholds, $Thres_{cold}$ and $Thres_{hot}$. If a large page's frequency value is smaller than $Thres_{cold}$, it is designated as cold. If its frequency value is greater than $Thres_{hot}$, it is a hot page. All other pages are designated as warm. We denote the gap between the two thresholds ($Thres_{hot} - Thres_{cold}$) as $length_{warm}$. Note that the state of warm is a transition one between the cold and hot states. We introduce it to avoid switching between the hot and cold states too often.

Identifying duplicate pages. We set a repetition rate threshold, $Thres_{repet}$, for the Selector to select candidate pages. In particular, the Selector only selects large pages whose percentages of duplicate subpages are more than $Thres_{repet}$ for splitting, and we name these pages as *duplicate large pages* or simply *duplicate pages*. It is important to set $Thres_{repet}$ properly so as to obtain a high deduplication rate with minimal number of split large pages. In our experiments, we find that by setting $Thres_{repet} = 1/8$, SmartMD can deduplicate more than 95% of duplicate subpages and split 40% fewer pages than traditional aggressive deduplication approach.

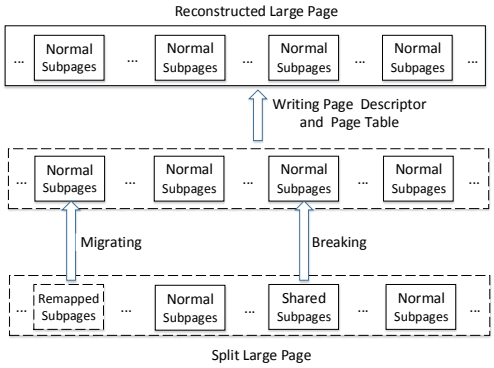


Figure 5: Process of reconstructing split large pages.

Selector Workflow. When scanning a large page, the Selector first reads its access frequency. If this page has been designated as cold, the Selector will further determine whether its repetition rate is greater than $Thres_{repet}$. If yes, this page is ready for splitting. On the other hand, when selecting split large pages for reconstruction, the Selector chooses only hot pages as candidates.

4.4 The Converter

The converter is responsible for the conversion between large pages and base pages, including the splitting of large pages and the reconstruction of split pages. The splitting process can be realized by calling a system API, while OSes do not well support the reconstruction functionality. We implement a utility in SmartMD to reconstruct split large pages. Fig. 5 illustrates this process, which consists of the following three steps.

(1) **Gathering subpages.** To reconstruct a split large page, we need to ensure that all of its subpages currently reside in memory and are not deduplicated with other pages. If some subpages have been deduplicated, we generate a duplicate copy for each of these subpages, and migrate all subpages to a contiguous memory space before reconstructing.

(2) **Writing page descriptor.** Once all subpages of a split large page have been gathered, we re-create page descriptor of the large page from the page descriptors of all subpages.

(3) **Writing page table.** We use a single page entry to map the reconstructed large page, and invalidate old entries about the original subpages.

As the cost of gathering subpages for reconstruction of large pages can be high, we propose two gathering mechanisms to reduce the number of subpages that have to be migrated. Specifically, if most of the subpages of a large page still stay in their original physical memory locations, we conduct *in-place gathering*, in which we migrate the subpages that have been relocated back to their original memory locations after migrating pages currently occupying the locations elsewhere. Otherwise, if most subpages of a split large page have been relocated from their original memory locations, we conduct *out-of-place gathering*, in which a contiguous memory space of the size of a large page is allocated and all of the large page's subpages are migrated into the space. Because of existence of spatial locality in the memory access, it is expected that for a particular workload either a high percentage of subpages of a split large page stay in the original locations or a high percentage of them do not. Our experiments show that for most benchmarks we tested, the percentages are larger than 90%. By adaptively applying the gathering mechanisms, we can significantly reduce gathering cost and the reconstruction overhead.

Adaptive page conversion. To reduce the cost of conversion between large pages and base pages, we develop an adaptive conversion scheme to improve performance of SmartMD based on the ratio of allocated memory size to total memory size, i.e., utilization of the memory space. The idea is that if the system has sufficient free memory space, we use only large pages for high memory access performance. On the other hand, if memory utilization becomes high and memory page swapping may occur, we split large pages into base pages for a high deduplication rate. Specifically, the adaptive page conversion scheme uses four parameters to guide its conversion decision, including two thresholds about memory utilization (mem_{low} and mem_{high}) and two thresholds about access frequency ($Thres_{cold}$ and $Thres_{hot}$). In each monitoring period, we first check the memory utilization, and then tune the parameter $Thres_{cold}$ accordingly so as to dynamically identify pages to be split. In particular, if the memory utilization is less than mem_{low}, we decrement $Thres_{cold}$ by one to make more pages stay in the warm or hot states and keep them from being split for high memory access performance. If the memory utilization is greater than mem_{high}, indicating that memory is in high demand, we increase $Thres_{cold}$ by one to allow more large pages to be considered as cold pages and be eligible for being split so as to achieve higher deduplication rate. Similar to a page's frequency value, we also keep $Thres_{cold}$ in the range from 0 to N, where N is a positive integer, in the process.

5 Evaluation

To show its efficacy and efficiency, we implement a SmartMD's prototype on Linux 3.4 and conduct experiments using QEMU to manage KVM. Our experiments run on a server with two Intel Xeon E5-2650 v4 2.20GHz processors, 64GB RAM, and a 2TB WD hard disk (WD20EFRX). Both the host and guest OSes are Ubuntu 14.04. We boot up four VMs in parallel, each of which is assigned one VCPU and 4GB RAM, and all VMs are hosted on one physical CPU. In our experiments, we focus only on 2MB and 4KB pages, which are commonly used in most applications. We run the following benchmark programs in each VM. Their memory demands without deduplication are listed in Table 3.

- **Graph500 [2].** Graph500 generates and compresses large graphs. It also runs breadth-first search on the graph. We run Graph500 in each guest VM with the same scale (22) and edgefactor (16). We generate graphs initialized differently to ensure that graphs in different VMs are different. We use average number of edges traversed in a VM per second as the performance metric of the benchmark.
- **SPECjbb [6].** SPECjbb is a benchmark for evaluating performance of server-side Java business applications. We run SPECjbb in each VM and use the average bops (business operations per second) of all VMs as its performance metric.
- **Libliner [5].** Libliner is a suite of linear classifiers for a data set with millions of instances and features. We run SVM, one benchmark program in Liblinear, on the urlcombined dataset. The performance metric is average execution time of the program running in different VMs.
- **Sysbench [7].** Sysbench is a multi-threaded benchmark for database. We run sysbench on Mysql by storing all data in the buffer pool of Mysql. We use the average number of queries performed by a VM per second as the performance metric.
- **Biobench [10].** Biobench is a suite of bioinformatics applications. We run Mummer, a program in Biobench on the human-chromosomes dataset [4], and measure its average execution time in different VMs.

Graph-500	SPECjbb-2005	Lib-linear	Sys-bench	Bio-bench
2.7GB	1.7GB	4.0GB	2.93GB	3.42GB

Table 3: Memory usage of each VM w/o deduplication.

We compare SmartMD with three other schemes on both performance and memory usage. The first one is

KSM, which uses the aggressive deduplication approach to split all large pages to achieve the best deduplication rate. The second one is named *no-splitting*, which preserves all large pages and performs deduplication in unit of large page to achieve the best access performance. The third one is Ingens [22], which is the state-of-the-art scheme using mixed pages to make a trade-off between access performance and memory saving. Default values of the parameters used in the experiments are listed in Table 4. We adopt the same rate at which for the schemes to scan and identify duplicate pages for a fair comparison.

Parameter	Value	Description
$monitor_period$	6s	scanning period of the monitoring thread
$check_interval$	2.6s	interv. of checking access bits
$Thres_{repet}$	1/8	thresh. of repetition rate
mem_{high}	90%	thresh. of high mem. utilization
mem_{low}	80%	thresh. of low mem. utilization
$page_to_scan$	1024	number of pages scanned by dedup-thread in each scan
$sleep_millisecs$	20ms	time to sleep after each scan of the dedup-thread

Table 4: Default parameter setting.

Note that with the adaptive page conversion scheme described in Section 4.4, large page will not be split for deduplication if there is a sufficient amount of free memory. In the evaluation of SmartMD on its effectiveness and efficiency (see Section 5.1∼5.4), we use fixed non-zero $Thresh_{cold}$ and $Thresh_{hot}$, instead of the adaptive conversion scheme, to make sure that SmartMD comes into effect even when the server has abundant free memory. Specifically, we set the range of a page's access frequency from 0 to 4. Meanwhile, instead of allowing $Thresh_{cold}$ to be decremented to 0 due to low memory utilization, we fix it at 1 so that large pages eligible for splitting may still be produced even if the system has enough free memory. In addition, we set $Thresh_{hot}$ to 3. We set initial access frequency of each page to 2, lying between $Thresh_{cold}$ and $Thresh_{hot}$, to ensure that it has a chance to be classified as either hot or cold page.

To evaluate effectiveness of the adaptive conversion scheme, we run experiments with SmartMD in a memory-constrained system (Section 5.5). In particular, we limit the host's memory space by running an in-memory file system (hugetlbfs [3]) to occupy a certain amount of memory space on the host. Pages held by hugetlbfs cannot be deduplicated or swapped out. In this way, we can flexibly adjust size of the host's memory available for running benchmark programs.

5.1 Overhead of SmartMD

CPU overhead. We first run Graph500 to compare the CPU overhead of SmartMD with the other two memory deduplication schemes, KSM and Ingens. The results are shown in Table 5. Both the monitoring thread and deduplication thread use additional CPU cycles. KSM uses aggressive deduplication without tracking status of the pages. However, without knowing whether a large page contains duplicate subpage(s), it has to scan all large pages and in each large page determines whether each of its subpages is a duplicate, leading to high CPU overhead in its deduplication. As shown in Table 5, KSM spends more CPU time than Ingens and SmartMD by 26% and 34%, respectively. SmartMD takes more CPU time on monitoring each large's access frequency and repetition rate. In contrast, Ingens monitors only access frequency. Accordingly, the monitoring thread of SmartMD induces 7.8% higher CPU overhead than that of Ingens. With the knowledge on access frequency and repetition rate of each large page, as well as on which of its subpage are duplicates, SmartMD can more efficiently and precisely locate large pages for effective deduplication. As shown Table 5, SmartMD's deduplication thread spends 9.4% lower CPU time than that of Ingens. Comparison of deduplication effectiveness with Ingens will be presented in Section 5.3.

	Monitor thread	Dedup thread	Total
KSM	0	33.5%	33.5%
Ingens	5.3%	21.3%	26.6%
SmartMD	13.1%	11.9%	25.0%

Table 5: Average CPU utilization sampled in every second.

Memory overhead. SmartMD uses 3 bits to store each of the eight counting bloom filters for each base page. Since the size of a base page is 4KB, the ratio of extra memory space used to store the filters is only $(3bits \times 8) \div (4KB) = 3/2^{12}$. For each large page, we use 32B to store its access frequency and repetition rate, as well as some necessary pointers. Since the size of a large page is 2MB, SmartMD requires additional $32B \div 2MB = 1/2^{16}$ of the memory space for large pages. For example, 16 GB memory is used during the running of Libliner on four VMs. SmartMD needs about 12MB to store bloom filers for base pages and 0.25MB to store metadata for large pages. Apparently the memory overhead of SmartMD is negligible.

5.2 Performance and Memory saving

In this section, we compare SmartMD with two commonly used mechanisms in major OSes, which are KSM or no-splitting, using different benchmark programs on their performance and memory usage. Comparison with Ingens will be presented in Section 5.3. By aggressively splitting any large pages to maximize deduplication opportunities, KSM can achieve the highest memory sav-

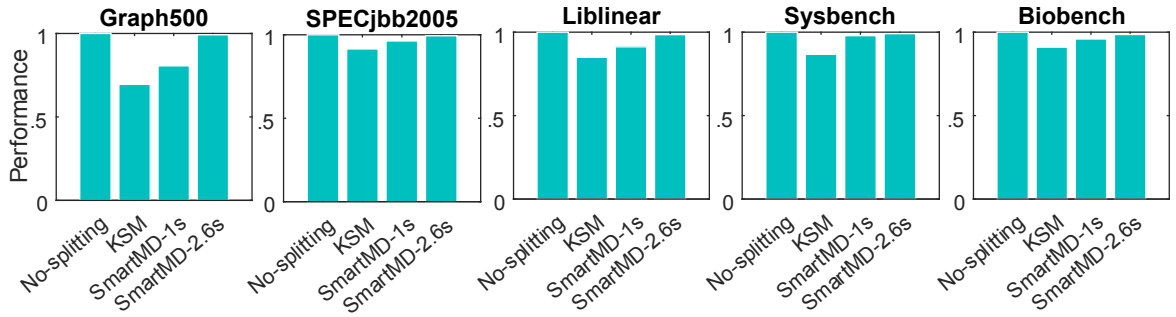


Figure 6: Performance of the benchmarks under various deduplication policies.

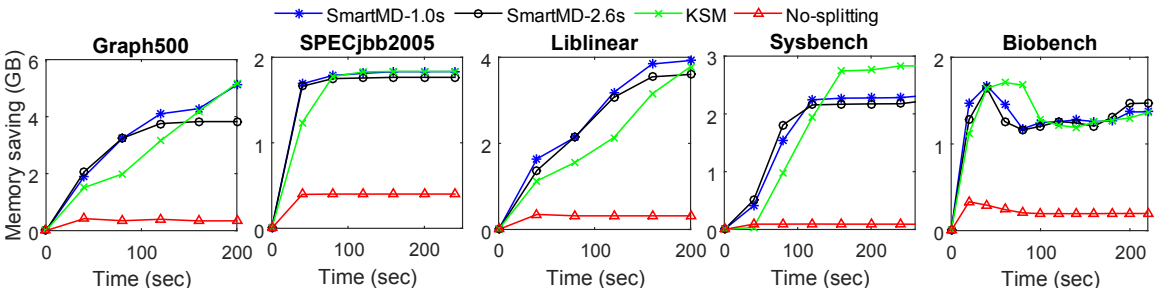


Figure 7: Memory saving under various deduplication policies.

ing. On the other hand, no-splitting represents an optimization only on performance by preserving all large pages. Here we study the trade-off made by SmartMD between performance and memory saving by comparing it with the KSM and no-splitting schemes.

We first show performance of the benchmarks by using SmartMD, KSM and no-splitting in Fig. 6, where we normalize the performance, whose metrics are introduced in the description of the benchmarks in Section 5, against that of the no-splitting scheme. In the experiments, we use two different check_interval values (1.0s and 2.6s) in SmartMD to vary the time period between reseting access bits and its next reaching of the bits. Accordingly, SmartMD is named SmartMD-1s and SmartMD-2.6s, respectively. Fig. 6 shows that for the benchmarks SmartMD achieves nearly the same performance as no-splitting by using a larger check_interval. In contrast, SmartMD improves KSM's performance by up to 42.7% by only spliting necessary large pages.

Experiment results on memory saving are shown in Fig. 7. Because no-splitting does not perform splitting of large pages and conducts deduplication in the unit of large page, it reduces memory usage by a small percentage (6% or less). In contrast, SmartMD and KSM can reduce memory usage by a much larger amount, which is usually 4x to 31x as large as the saving received in no-splitting. Fig. 7 also show , we can also see that in general SmartMD reduces about the same amount of memory as KSM. Interestingly, in some execution periods of some benchmarks, such as Liblinear, SmartMD reduces more memory than KSM. By using counting Bloom filters and labeling of duplicate pages, SmartMD can complete its scan of memory to find duplicate pages much faster than KSM, and carry out deduplication in a more timely manner. For example, to reduce memory usage of Liblinear by 3.2GB SmartMD-2.6s and KSM take 118s and 161s, respectively.

Looking into Figs. 6 and 7, we can see that SmartMD takes both benefits on memory saving and access performance. Specifically, SmartMD can save 4x to 21x as much memory as the no-splitting scheme while keeping similar access performance. For example, with Graph500 SmartMD can save 3.82 GB memory space, or 35.4% of the total memory, which is 9x the memory space saved by no-splitting. In the meantime, SmartMD can achieve up to 15.8% of performance improvement over KSM while achieving a memory saving similar to KSM.

Additionally, SmartMD can be configured to tune the weight of its optimization goals between access performance and memory saving. With SmartMD, we can improve either the access performance or memory saving while minimally compromising the other goal. For example, the performance of Sysbench is improved by 12.9% with increasing checking interval from 1.0s to 2.6s. Meanwhile, the memory saving only decreases by 4.3%. This is because SmartMD splits only large pages with low access frequency and high repetition rate. In this way, SmartMD can ensure that each splitting can bring benefit of memory saving but incur small negative impact on memory access performance. Furthermore, base pages can be opportunistically converted back to large pages to benefit the performance of SmartMD.

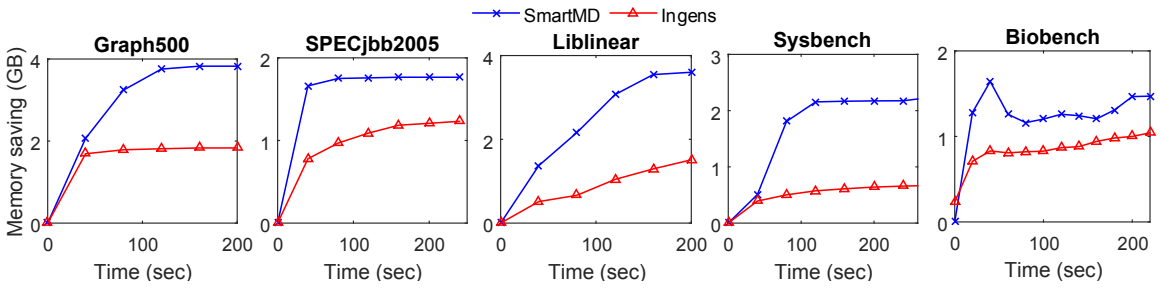


Figure 8: Comparison of memory saving between Ingens and SmartMD.

	Ingens	SmartMD
Graph500	0.989	0.992
SPECjbb2005	0.991	0.994
Liblinear	0.987	0.991
Sysbench	0.982	0.989
Biobench	0.976	0.982

Table 6: Performance normalized to that of No-splitting.

	Single-CPU	NUMA
Graph500	0.8%	1.6%
SPECjbb2005	0.6%	2.1%
Liblinear	0.9%	1.8%
Sysbench	1.1%	2.6%
Biobench	1.8%	3.9%

Table 7: Performance degradation by using SmartMD on NUMA. The degradation is calculated against the performance of No-splitting with the same benchmark.

5.3 Comparison with Ingens

Kwon et al. [22] proposed Ingens to enable conversion from base pages to large pages to maintain high memory access. It also selectively splits large pages for more effective deduplication. However, in the selection of large pages, it only considers access frequency and does not take into account of repetition rate. In addition, it does not consider page access frequency in the decision of reconstruction of large pages. Table 6 shows the performance of the benchmark programs with SmartMD and Ingens. For SmartMD check_interval is set at 2.6s. Fig. 8 shows the memory saving of SmartMD and Ingens. We see that SmartMD can save 1.3x to 3.5x as much memory as Ingens while still keeping performance of SmartMD to that of Ingens. While Ingens splits any large pages that are considered cold, it has to throttle generation of cold pages to keep memory access performance close to that of no-splitting. This is achieved by postponement of checking accessing bits. However, this approaches leaves fewer pages available for deduplication. SmartMD can more precisely identify the right large pages (with low access frequency and high repetion rate) for splitting. It is less likely to conduct unnecessary splitting. SmartMD also performs necessary reconstruction of large pages to keep high memory performance.

5.4 Performance in a NUMA Environment

In the above experiments, all VMs are hosted on one physical CPU in a NUMA system. However, if they are hosted on different CPUs, deduplication may make accesses of originally local pages become more expensive ones of remote pages, causing performance degradation.

To study performance impact of the NUMA architecture, we place two VMs on one physical CPU, and another two on a different CPU and re-run the benchmarks with SmartMD. The performance results are shown in Table 7. As shown, running SmartMD in the NUMA environment does cause larger performance degradation. However, the NUMA impact is very small, as SmartMD only splits large pages into base pages and deduplicate them only for those with low access frequency. Thus, even if many pages are deduplicated and relocated, only a very limited number of remote accesses are induced.

5.5 Performance in a Memory Overcommitted System

In this section, we evaluate the performance with different memory loads: no-overcommitted, slight-overcommitted and severe-overcommitted, which correspond to scenarios where the ratios of memory demand of an application to the usable memory size as 0.8, 1.1, and 1.4, respectively. We compare the performance of benchmarks using KSM, Ingens, and No-splitting, and the performance results are shown in Fig. 9. We can see that when the system has sufficient memory, performance of SmartMD is close to that of No-splitting. This is because when the memory utilization is low, SmartMD sets the cold threshold ($Thresh_{cold}$) to zero to keep large pages from being split.

With the increase of the host's memory load, the access performance of No-splitting drops much faster than other three schemes. With less effective deduplication, No-splitting has a larger memory demand. When the demands is larger than usable memory size, it will cause

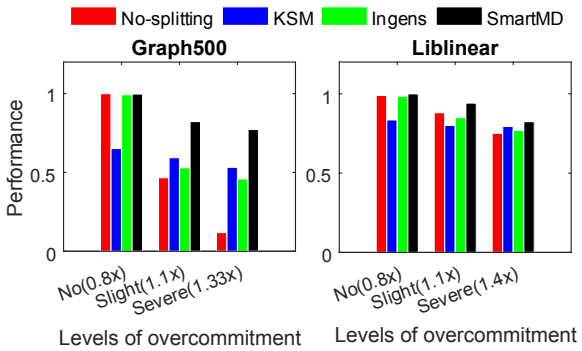


Figure 9: Performance in overcommitted systems.

more serious swapping of the program's working set between the memory and the disk, significantly slowing down the program's execution. With few pages deduplicated and larger memory demand than SmartMD, Ingens also shows significantly degraded performance in a memory overcommitted system.

SmartMD outperforms the other three schemes in the memory overcommitted systems. For example, for Graph500 SmartMD achieves up to 38.6% of performance improvement over other schemes. Using intelligently selective and adaptive conversion between large pages and base pages, SmartMD can make a better tradeoff between memory saving and access performance under different levels of memory overcommitments.

6 Related Work

Management of large pages. To efficiently use large pages, researchers proposed schemes to manage pages of different sizes [16, 26]. For example, Navarro et al. [25] provide a tool for FreeBSD to support multiple page sizes with contiguity-awareness and fragmentation reduction. Gorman et al. [18] propose a placement policy for physical page allocator, which mitigates fragmentation and increases contiguity by grouping pages according to whether the pages can be migrated. Their subsequent work [19] proposes an API for applications to explicitly request huge pages. Different from SmartMD, the above works do not consider memory deduplication.

Memory Deduplication. Memory deduplication has attracted attention of many researchers [11, 24, 20, 29, 28, 15, 21, 30]. In-memory deduplication technique was first implemented in VMWares ESX server [30], which requires no assistance from guest OSes and performs transparently in the hypervisor layer. KSM [11] is implemented as a kernel thread, which periodically scans memory pages to detect duplicate pages. Miller et al. [24] find that data in the page cache are more likely to be duplicates. They propose a memory deduplication scanner named XLH to identify duplicate pages. Gupta et al. propose Difference Engine [20] to deduplicate partial base pages with partial redundancy. The above works can be considered as aggressive deduplication schemes whose sole objective is to reduce memory usage. However, they do not consider impact of using large pages on deduplication efficacy as well as performance impact of splitting large pages.

The Ingens Deduplication Ingens [22] is a recently proposed memory deduplication scheme most similar to SmartMD. Ingens provides a coordinated transparent huge page support for the OS and hypervisor. In contrast, SmartMD achieves higher memory saving while maintaining similar access performance with its three advantages. (1) SmartMD selectively splits large pages according to their access frequency and repetition rate, while Ingens only considers pages' access frequency. (2) SmartMD reconstructs split large pages based on their access frequency, while Ingens reconstructs a large page as long as most of its subpages are utilized. (3) SmartMD adaptively selects pages for splitting and reconstruction, and uses sampling-based counting bloom filters and duplication labels to reduce overhead.

7 Conclusion

In this work, we propose SmartMD, an adaptive and efficient scheme, to manage memory with pages of different sizes. SmartMD can simultaneously take both the benefit of high performance by accessing memory with large pages, and the benefit of high deduplication rate by managing memory with base pages. Experimental results show that compared to KSM and no-splitting, SmartMD can either saves more memory space with similar memory access performance, or achieves higher memory access performance with similar memory saving.

8 Acknowledgements

We thank the anonymous reviewers and our shepherd, Don Porter, for their valuable comments and suggestions. In the work, Yongkun Li was partially supported by Anhui Provincial Natural Science Foundation (1508085SQF214). Yinlong Xu was partially supported by National Natural Science Foundation of China (61379038). Song Jiang was partially supported by US National Science Foundation (1527076). John C. S. Lui was partially supported by Hong Kong General Research Fund (14208816). Yongkun Li is the corresponding author.

References

[1] bloomfilter. https://en.wikipedia.org/wiki/Bloom_filter.

[2] Graph500. http://www.graph500.org/specifications.

[3] Hugetlbfs. https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt.

[4] Human chromosomes. http://mummer.sourceforge.net/applications.html.

[5] Liblinear. https://www.csie.ntu.edu.tw/~cjlin/liblinear/.

[6] SPECjbb2005. https://www.spec.org/jbb2005/.

[7] Sysbench. https://github.com/akopytov/sysbench.

[8] The big khugepaged redesign. https://lwn.net/Articles/634384/.

[9] Page sizes among architectures. https://en.wikipedia.org/wiki/Page_(computer_memory), 2017.

[10] ALBAYRAKTAROGLU, K., JALEEL, A., WU, X., FRANKLIN, M., JACOB, B., TSENG, C.-W., AND YEUNG, D. Biobench: A benchmark suite of bioinformatics applications. In *ISPASS* (2005), IEEE.

[11] ARCANGELI, A., EIDUS, I., AND WRIGHT, C. Increasing memory density by using ksm. In *Proceedings of the linux symposium* (2009), Citeseer, pp. 19–28.

[12] BHARGAVA, R., SEREBRIN, B., SPADINI, F., AND MANNE, S. Accelerating two-dimensional page walks for virtualized systems. In *ACM SIGARCH Computer Architecture News* (2008), vol. 36, ACM, pp. 26–35.

[13] BUELL, J., HECHT, D., HEO, J., SALADI, K., AND TAHERI, R. Methodology for performance analysis of vmware vsphere under tier-1 applications. *VMware Technical Journal 2*, 1 (2013), 19–28.

[14] CHANG, C.-R., WU, J.-J., AND LIU, P. An empirical study on memory sharing of virtual machines for server consolidation. In *ISPA* (2011), IEEE, pp. 244–249.

[15] CHIANG, J.-H., LI, H.-L., AND CHIUEH, T.-C. Introspection-based memory de-duplication and migration. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 51–62.

[16] FANG, Z., ZHANG, L., CARTER, J. B., HSIEH, W. C., AND MCKEE, S. A. Reevaluating online superpage promotion with hardware support. In *HPCA* (2001), IEEE.

[17] GANDHI, J., BASU, A., HILL, M. D., AND SWIFT, M. M. Efficient memory virtualization: Reducing dimensionality of nested page walks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (2014), IEEE Computer Society, pp. 178–189.

[18] GORMAN, M., AND HEALY, P. Supporting superpage allocation without additional hardware support. In *Proceedings of the 7th international symposium on Memory management* (2008), ACM, pp. 41–50.

[19] GORMAN, M., AND HEALY, P. Performance characteristics of explicit superpage support. In *International Symposium on Computer Architecture* (2010), Springer, pp. 293–310.

[20] 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.

[21] KRUUS, E., UNGUREANU, C., AND DUBNICKI, C. Bimodal content defined chunking for backup streams. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies* (2010), FAST'10.

[22] KWON, Y., YU, H., PETER, S., ROSSBACH, C. J., AND WITCHEL, E. Coordinated and efficient huge page management with ingens. In *OSDI 16* (2016), USENIX Association, pp. 705–721.

[23] MERRIFIELD, T., AND TAHERI, H. R. Performance implications of extended page tables on virtualized x86 processors. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2016), VEE '16, ACM.

[24] MILLER, K., FRANZ, F., RITTINGHAUS, M., HILLENBRAND, M., AND BELLOSA, F. Xlh: More effective memory deduplication scanners through cross-layer hints. In *USENIX Annual Technical Conference* (2013), pp. 279–290.

[25] NAVARRO, J., IYER, S., DRUSCHEL, P., AND COX, A. Practical, transparent operating system support for superpages. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 89–104.

[26] NAVARRO, J. E. *Transparent operating system support for superpages*. PhD thesis, Rice University, 2004.

[27] PHAM, B., VESELỲ, J., LOH, G. H., AND BHATTACHARJEE, A. Large pages and lightweight memory management in virtualized environments: Can you have it both ways? In *Proceedings of the 48th International Symposium on Micro architecture* (2015), ACM, pp. 1–12.

[28] SHARMA, P., AND KULKARNI, P. Singleton: system-wide page deduplication in virtual environments. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing* (2012), ACM, pp. 15–26.

[29] SINDELAR, M., SITARAMAN, R. K., AND SHENOY, P. Sharing-aware algorithms for virtual machine colocation. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures* (2011), ACM, pp. 367–378.

[30] WALDSPURGER, C. A. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 181–194.

Elastic Memory Management for Cloud Data Analytics

Jingjing Wang and Magdalena Balazinska
Dept. of Computer Science & Engineering, University of Washington

Abstract

We develop an approach for the automatic and elastic management of memory in shared clusters executing data analytics applications. Our approach, called ElasticMem, comprises a technique for dynamically changing memory limits in Java virtual machines, models to predict memory usage and garbage collection cost, and a scheduling algorithm that dynamically reallocates memory between applications. Experiments with our prototype implementation show that our approach outperforms static memory allocation leading to fewer query failures when memory is scarce, up to 80% lower garbage collection overheads, and up to 30% lower query times when memory is abundant.

1 Introduction

The analysis of large datasets is an important problem and many big data systems are available to facilitate this task [2,29,33,36,48,53]. To handle large data sizes, these systems execute in shared-nothing clusters. Whether public or private, clusters are typically shared by many queries (also called "applications")[1] and even many systems executing in the same cluster at the same time. In such shared clusters, a resource manager [25,47] is responsible for the resource allocation between systems and applications. Modern resource managers rely on containers (*e.g.*, YARN [47], Docker [3], or Kubernetes [5] containers), which isolate applications that share the same machine and provide hard resource limits. Application resource requirements are both constrained and protected by the containers. Figure 1 illustrates the interaction between a resource manager and containers: the resource manager launches containers with resource limits and schedules applications inside those containers.

Many modern data analytics systems, such as Spark [53], Impala [29], GraphLab [33], Giraph [2], and Myria [22,48], strive to maximally utilize memory, yet memory remains an expensive resource. In this paper, we focus in particular on *memory* allocation. However, container-based scheduling has limitations for managing memory. When an application needs to run, it must estimate its resource requirements and communicate them to the resource manager. The latter then decides whether or not to schedule the application based on the amount of available resources. The challenge, however, is that it is hard to estimate the memory need of a data analytics application before executing it because it may depend on multiple runtime factors including the cardinalities of intermediate results, which are known to be hard to estimate [27,31].

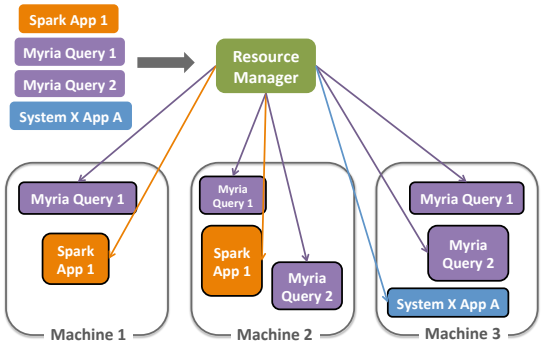


Figure 1: A resource manager schedules multiple applications from multiple systems (Spark [53], Myria [48], and System X) in a shared cluster. An application may have multiple processes across multiple machines. The resource manager schedules applications by putting them in containers with resource limits.

Having an inaccurate memory usage estimate can harm query performance in multiple ways. If the estimate is too high, cluster resources may be under-utilized. If the estimate is less than the minimum amount of memory needed to complete the query, the system must either spill data to disk, which leads to performance degradation, or fail with an out-of-memory error, wasting the resources

[1] In this paper, we focus on applications that correspond to analytical queries and use the terms "application" and "query" interchangeably.

already consumed by the query. This challenge exists in systems with manual memory management, such as those written in C/C++ [29, 33], in Java-based systems that use byte arrays [8], and in systems that rely on automatic memory management provided by runtimes such as Java [2, 48, 52, 53] and the .NET Common Language Runtime (CLR) [36]. The situation is more complicated when garbage collection (GC) is used for automatic memory management, since GC activities add another layer of unpredictability to query performance. Even if the resource estimates are sufficient for the query to complete, garbage collection in some cases can significantly slow down query execution. As a concrete example, we demonstrate how changing the maximum heap size of Java-based systems can significantly impact query time in Section 2.

To address these problems, we develop a new approach, called *ElasticMem*, where data analytics applications execute in separate containers, but the resource manager *elastically* adjusts the memory allocated to these containers. The optimization goal is to jointly minimize failures and total execution time of all applications subject to the physical limit on the total amount of memory in the cluster. We presented the vision behind the approach and a few preliminary results in a short workshop paper [49]. In this paper, we develop the approach in full.

Elastic container memory management is a difficult problem. First, elastic memory allocation is not supported in most systems. For Java-based systems, the maximum heap size of a Java virtual machine (JVM) stays constant during its lifetime. For C/C++-based systems such as Impala [29], limiting the resource of a process is usually done through Linux utilities such as `cgroups`, which do not expose functionality to change resource limits at runtime. For systems that run in CLR [36], the problem is opposite: No control on the heap size can be specified, so the heap can grow arbitrarily up to the total physical memory. Second, in order to elastically and dynamically allocate memory to data analytics applications, we need to understand how extra memory can prevent failures and speed up these applications. We need models of GC benefits and overheads. Finally, we need an algorithm that uses the models to orchestrate memory allocation across multiple data analytics applications.

We present our approach to address all three challenges. We focus on analytical applications, in particular *relational algebra* queries on large data, and Java-based systems. Since memory management in Java containers (e.g., YARN [47]) is determined by JVMs internally, we focus on how and when to change the memory layouts of JVMs. Specifically, our contributions are the following:

- We show how to modify the JVM to enable dynamic changes to an application's heap layout for elastic management of its memory utilizations (Section 3.1).
- Our key contribution is an algorithm for elastically managing memory across multiple applications in a big data analytics system to achieve an overall optimization goal (Section 3.2). In this paper, we present scenarios where each query runs in one JVM and multiple queries run in one machine, but our approach can be extended to a multi-machine setting.
- In support of elastic memory management, we develop a machine-learning based technique for predicting the heap state and GC overhead for a relational query and whether it is expected to run out of memory (Section 3.3) based on operator statistics. Since the common approach for implementing relational operators in memory, such as joins and aggregates, is to use hash tables [19], we build models that use hash table statistics as input.

We evaluate our elastic memory management techniques using TPC-H queries [6] on Myria [22, 48], a shared-nothing data analytics system, against containers with fixed memory limits. In our experiments, our approach outperforms static allocation: It reduces the number of query failures; it reduces query times by up to 30%, GC times by up to 80%, and overall resource utilization (Section 4).

2 Performance Impact of Automatic Memory Management

Many big data analytics systems today, including Spark [53], Flink [1], Hadoop [52], Giraph [2], and Myria [48], are written in programming languages with automatic memory management, specifically Java. Garbage collection associated with automatic memory management is known to cause performance variations that are hard to control: The GC policy, although customizable by the programmer to some extent, is controlled by the runtime internally. Depending on the policy and heap state, the time and frequency of GCs may vary significantly and, as we later show in this section, may significantly impact query performance.

Over the past decade, there have been several JVM implementations with various GC algorithms. However, most of the contemporary ones share the concept of generations [9]. With this design, the heap space is partitioned into multiple generations for storing objects with different ages. Figure 2 illustrates the internal state of a JVM heap with two generations. Initial memory allocation requests always go to the young generation. When it fills up, a GC is triggered to clean up dead objects. There are different types of GCs as shown in Figure 2. In a young collection, live objects in the young generation are promoted to the old generation. In a full collection, dead objects are cleaned from both generations in addition to promotions. The type of collection to trigger depends on whether a promotion failure, *i.e.*, insufficient space for promoting

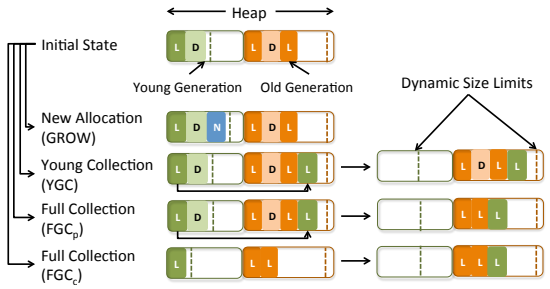


Figure 2: Internal heap states of a JVM before and after actions of new object allocation, young generation collection, and full collections, starting from an initial state. Dark blocks: (L)ive objects, light blocks: (D)ead objects, blue blocks: (N)ew objects. Dashed lines: generation size limits that can be changed in real time by our approach. We describe `FGC`$_p$ and `FGC`$_c$ in Section 3.2.3.

objects from the young generation, is expected to occur or actually occurs. In this paper, we use OpenJDK as the reference JVM implementation. We focus on the common class of GC algorithms that use a young and old generation, and leave extensions to other languages and GC algorithms to future work.

We show a concrete example of how GC can impact query execution by executing a self-join query on a synthetic dataset containing ten million tuples with two `int` columns, on three systems: Myria, Spark 1.1 and Spark 2.0, using one process on one machine with default GC collectors (`-XX:+UseParallelGC`). Figure 3 shows the query execution times with different heap-size limits. Each data point is the average of five trials with error bars showing the minimum and maximum values. For both Myria and Spark 2.0, when the heap is large, the query time converges to approximately 35 seconds, which is the pure query time with almost no GC. When we shrink the heap size, however, the run times increase moderately due to more GC time. For Myria, the run time increases from 35 seconds to 55 seconds when the heap size goes from 16 GB to 3 GB, and further increases drastically to 141 seconds when the heap size shrinks from 3 GB to 2 GB. Eventually, Myria fails with an out-of-memory error when the limit is less than 2 GB. Similarly, the query time for Spark 1.1 has a steep increase from 86 to 466 seconds when the heap size changes from 5 to 4 GB, and the query fails when the heap size is less than 4 GB. Spark 2.0 follows a similar trend as Myria, but does not fail even with only 500 MB of memory because it is able to spill data to disk when memory is insufficient. As a result, however, its execution time increases to 127 seconds.

3 Elastic Memory Allocation

In this section, we present our approach, called *ElasticMem*, for elastic memory allocation. ElasticMem comprises three key components. First, ElasticMem needs

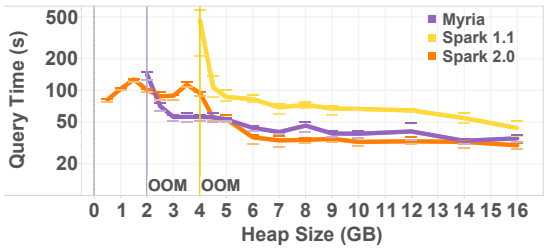


Figure 3: Impact of GC on query execution time in Myria, Spark 1.1, and Spark 2.0. The y-axis uses a log scale.

JVMs that can change memory limits dynamically, and we describe how we modify OpenJDK to enable this feature in Section 3.1. Second, the heart of ElasticMem is a memory manager that dynamically allocates memory across multiple queries (Section 3.2). Finally, to drive the manager's allocation decisions, ElasticMem uses models that predict the heap state and the GC costs (*i.e.*, impact on run time) and benefits (*i.e.*, expected freed memory) at any point during query execution (Section 3.3). The implementation of our approach is available at our website [4].

3.1 Implementing Dynamic Heap Adjustment in a JVM

OpenJDK manages an application's memory as follows: First, the user specifies the maximum heap size of a JVM process before launching it. The JVM then asks the operating system to reserve the heap space and divides the space into generations based on its internal size policy as in Figure 2. During program execution, if a memory allocation request cannot be satisfied due to insufficient memory, the JVM may trigger GCs to release some memory. If not much memory is released after spending a large amount of time on GC, the JVM throws an `OutOfMemory` error. The maximum heap size stays constant during a JVM's lifetime. It cannot be increased even if an `OutOfMemory` is thrown while more memory is available on the machine, or decreased if heap space is underutilized.

This rigid design, however, is unnecessary. For operating systems that support overcommitting memory, a logical address space does not physically occupy any memory until it is used. This property, together with 64-bit address spaces, allow us to reserve and commit a large address space when launching a JVM. The actual memory limits on heap spaces, such as generations, can be modified later during runtime.

We modify the source code of OpenJDK to implement this feature. We change the JVM to reserve and commit a continuous address space of a specified maximum heap size (`-Xmx`) when it launches. The initial size limit of each generation is set according to the JVM's internal policy. We make the maximum heap size large enough such that the per-generation limits are sufficiently large to

become irrelevant. Additionally, we add our new dynamic size limits to both the young and old generation of a JVM p, denoted with $y_{limit}(p)$ and $o_{limit}(p)$ respectively. Our memory manager changes these limits at runtime. We set their initial values to reasonably small numbers (*e.g.*, 1 GB) and prevent each generation from using more memory than its dynamic limit.

To interact with the JVM, we add a socket-based API through which the JVM receives instructions such as requests for the current heap state, memory limit adjustments, or GC triggers. We disable the JVM's internal GC policies to let our memory manager control when and which GCs to happen. We modify GC implementations to always release recycled memory to the OS. If more memory is needed but unavailable given the current limits, we let the JVM pause until more memory is available. We implement our changes on top of OpenJDK 7u85's default heap implementation (`ParallelScavengeHeap`), which contains approximately 1000 lines of code.

3.2 Dynamic Memory Allocation

The main component of ElasticMem is a memory manager. It monitors concurrently executing queries and alters their JVMs' memory utilizations by performing actions on the JVMs, such as triggering a GC or killing the JVM. Each action has a value, and the objective is to maximize the sum of all action values. A value is a combination of several factors, including whether the action kills a JVM, causes a JVM to pause, or how efficiently it enables the JVM to acquire memory: *i.e.*, the ratio of time spent over space acquired (from the OS or recovered through a GC).

The manager makes decisions according to two pieces of information: the JVM heap states and the estimated values of performing actions on the JVMs. Because predicting these values far into the future carries significant uncertainty, and because our changes to the JVM enable us to adjust memory limits without any overhead, we develop a dynamic memory manager. The manager makes decisions adaptively at each timestep t for some small period $[t, t+\delta_t]$. At t, the manager gathers runtime statistics from each JVM and performs actions on it. Queries then execute for time δ_t. Their states change and the manager makes another round of decisions at $t+\delta_t$. We describe our allocation algorithms in this section, starting with a more precise problem statement.

3.2.1 Problem Statement

We start with a single-node and a one-process-per-query scenario. As introduced in Section 1, each JVM is a container that executes a single query (or query partition). We model query execution as the process of accommodating the memory growth of the corresponding JVM. For a period $[t, t+\delta_t]$, the memory usage of a JVM may grow by some amount. We can perform various actions to the JVM to affect its memory utilization: allocate enough memory for the expected growth, trigger a GC, which may require extra memory in the short term but free up memory in the longer term, kill the JVM to release all its memory, or do nothing, which may stall a JVM if it cannot grow its memory utilization as needed.

Consider a single physical machine with a total amount of memory M. A set of N JVMs $\{p_1, \ldots, p_N\}$ is running on it, each has used some space in both the young and the old generation. At the current timestep t, we need to allocate M across the N JVMs, such that the total memory used does not exceed M, while minimizing a global objective function.

The memory that must be allocated to a JVM is entirely determined by the action that the manager selects. For example, to perform a young generation GC, the old generation needs to have enough space to accommodate the promoted young generation live objects. The manager must increase the memory limit for the old generation to accommodate the added space requirement. We denote with $y_{cap}(p_i, a_i)$ and $o_{cap}(p_i, a_i)$, the minimal amount of memory that must be allocated to the young and old generation of JVM p_i, if the manager chooses action a_i. These values refer to the new required totals and not increments.

Each action has a value that contributes to the global objective function. We denote the value of action a_i on p_i with $value(p_i, a_i)$. The objective function is thus:

$$\begin{aligned} &\text{maximize} && \sum_{i=1}^{N} value(p_i, a_i), a_i \in Actions, \\ &\text{subject to} && \sum_{i=1}^{N} (y_{cap}(p_i, a_i) + o_{cap}(p_i, a_i)) \leq M, \end{aligned}$$

where *Actions* is the set of possible actions. In our approach, the $value(p_i, a_i)$ is a structure with multiple fields. We describe its internal structure and how to sum and compare values in Section 3.2.3 below.

The above definition can be extended to a shared-nothing cluster scenario by letting the manager make decisions independently for each machine.

3.2.2 Runtime Metrics

Several runtime metrics are needed to compute the value and the space requirements of actions. Some are reported by the JVM while others are estimated by the manager:

Metrics reported by the JVM: For a JVM p at timestep t, $y_{limit}(p,t)$ and $o_{limit}(p,t)$ are the current memory limits of the young and old generation. The manager sets those limits at the previous timestep. However, only some of the space in each generation is used at t, and the JVM reports the used sizes as $y_{used}(p,t)$ and $o_{used}(p,t)$.

Metrics estimated by the manager: Besides the above metrics, we also need to estimate some values

Value	Meaning
$y_{limit}(p,t)$	Size limit of the young gen
$o_{limit}(p,t)$	Size limit of the old gen
$y_{used}(p,t)$	Total used space in the young gen
$o_{used}(p,t)$	Total used space in the old gen
$\hat{y}_{live}(p,t)$	Total size of live objects in the young gen
$\hat{o}_{live}(p,t)$	Total size of live objects in the old gen
$\hat{y}_{dead}(p,t)$	Total size of dead objects in the young gen
$\hat{o}_{dead}(p,t)$	Total size of dead objects in the old gen
$\hat{grw}(p,t)$	Estimated heap growth until next timestep
$\hat{gc}_y(y_{obj}(p,t))$	Time to perform a young collection
$\hat{gc}_o(o_{obj}(p,t))$	Time to perform an old collection

Table 1: Runtime metrics reported by JVM p or estimated by the manager at timestep t. $\hat{x}$ indicates that x is estimated. "gen" is short for generation.

that are not directly available. First, the space used in the young and old generation of a JVM is further divided into live and dead objects. The manager estimates the total size of those objects, which we denote with $\hat{y}_{live}(p,t)$, $\hat{y}_{dead}(p,t)$, $\hat{o}_{live}(p,t)$ and $\hat{o}_{dead}(p,t)$. We use $\hat{x}$ to indicate that a value x is estimated by the manager. Second, the manager needs to estimate p's heap growth, $\hat{grw}(p,t)$, before the next timestep, where $\hat{grw}(p,t) = \hat{y}_{used}(p,t+\delta_t) - y_{used}(p,t)$. Finally, to model the impact of a GC, the manager needs to know how much memory a GC will free, and how much time it will take. Since the target of a GC is the set of all objects in the generation(s) undergoing the GC, we use $y_{obj}(p,t)$ to denote the set of all the objects in the young generation and similarly $o_{obj}(p,t)$ for the old generation.[2] $\hat{gc}_y(y_{obj}(p,t))$ and $\hat{gc}_o(o_{obj}(p,t))$ are then the estimated times for a young and an old GC. We describe how the manager estimates these metrics in Section 3.3.

Table 1 summarizes the notation. Since t is the only used timestep, we omit t and only use p as the argument in the rest of the paper when the context is clear.

3.2.3 Space of Possible Actions

There are four types of actions that the manager can choose for each JVM: allowing the JVM to grow by asking the operating system for more memory, reducing the memory assigned to the JVM by performing a garbage collection and recycling space,[3] pausing the JVM if it cannot either grow or recycle enough memory, or as a last resort, killing a JVM to release its entire memory. The manager performs an action for every JVM at each timestep. An action a on a JVM p has value, $value(p,a)$, with a minimum amount of memory needed for p's young and old generations, ($y_{cap}(p,a)$ and $o_{cap}(p,a)$). We denote the time to perform a on p with $time(p,a)$, and the size of the newly available space made by a with $space(p,a)$. The *cost* of an action is the amount of time needed to acquire a given amount of space, or $\frac{time(p,a)}{space(p,a)}$. The manager uses this ratio to compare and choose actions.

The detailed set of *Actions* is as follows:

- `GROW`: Let the JVM grow to continue query execution. In order to reserve space for the growth, the manager must allocate $y_{cap}(p,\texttt{GROW}) = y_{used}(p) + \hat{grw}(p)$ to the young generation and $o_{cap}(p,\texttt{GROW}) = \hat{y}_{live}(p) + o_{used}(p)$ to the old generation. We reserve extra space in the old generation for prospective promotions to preserve the possibilities of having all types of GCs in the future. The cost is the time it takes to request and access the new space, which depends on the size of the space change given by: $y_{cap}(p,\texttt{GROW}) + o_{cap}(p,\texttt{GROW}) - y_{limit}(p) - o_{limit}(p)$. Under normal circumstances, this will be the commonly selected action until space becomes tight and JVMs must start garbage collection or must pause before being able to grow again.
- `YGC`: Trigger a young generation GC. The JVM needs at least the current used space, $y_{used}(p)$, for the young generation, and $\hat{y}_{live}(p) + o_{used}(p)$ for the old generation to avoid a promotion failure. The cost is the GC time $\hat{gc}_y(y_{obj}(p))$, and we expect memory of size $\hat{y}_{dead}(p)$ to be recycled.
- $\texttt{FGC}_\texttt{p}$: Trigger a full GC by first performing a young generation collection to promote live objects to the old generation then performing a GC on the old generation. Similar to `YGC`, we need at least $y_{used}(p)$ and $\hat{y}_{live}(p) + o_{used}(p)$ for the young and old generations respectively. The cost is the GC time $\hat{gc}_y(y_{obj}(p)) + \hat{gc}_o(o_{obj}(p))$ and the space to be recycled is $\hat{y}_{dead}(p) + \hat{o}_{dead}(p)$.
- $\texttt{FGC}_\texttt{c}$: Trigger a full GC by first performing a GC on the whole heap, then trying to promote young generation live objects if possible, without changing the total heap size. Free space from the young generation after the first GC gets shifted to the old generation to make space for copying. Different from $\texttt{FGC}_\texttt{p}$, we only need $y_{used}(p)$ and $o_{used}(p)$ for the young and old generation since the promotion is not mandatory. However, more GC time is needed since the full collection is now performed on both generations instead of only the old generation. We assume that the time grows proportionally to the size of live objects and use $\hat{gc}_y(y_{obj}(p)) + \hat{gc}_o(o_{obj}(p)) * (\hat{y}_{live}(p) + \hat{o}_{live}(p))/\hat{o}_{live(p)}$ as the GC time estimate. The memory to be recycled is also $\hat{y}_{dead}(p) + \hat{o}_{dead}(p)$.
- `NOOP`: Do nothing to the JVM, keep the current limits $y_{limit}(p)$ and $o_{limit}(p)$. As a consequence, the JVM is expected to pause since it cannot either grow or recycle enough memory by doing garbage collection.
- `KILL`: Kill the JVM immediately. As a consequence, the query running in this JVM will fail.

$\texttt{FGC}_\texttt{p}$, which promotes first, is the default behavior in

[2] $y_{obj}(p,t)$ is the union of all the live and dead objects in the young generation of p at t, similarly to $o_{obj}(p,t)$.

[3] The recycled memory is always reclaimed by the OS.

OpenJDK. However, the promotion may fail if the old generation does not have enough free space to absorb young generation live objects, and when it happens, JVM spends much time on copying the live objects back, so that the young generation remains the same as it was before the promotion. In other words, triggering $\texttt{FGC}_\texttt{p}$ while expecting a promotion failure is not cost effective. However, when memory is scarce, the manager may not be able to allocate extra space to avoid the promotion failure. In this case, we need a GC which can still recycle space without increasing the limits. We solve this problem by implementing another full GC procedure, $\texttt{FGC}_\texttt{c}$: We first collect both generations, then shift young generation free space to the old generation to keep the total heap limit unchanged. A young GC is then performed if there is enough space for promoting.

Table 2 summarizes the properties of all actions. $os(m)$ denotes the time to access new memory of size m. We obtain its value by running a calibration program since this value changes for different systems and settings. Figure 2 illustrates the effect of all the actions except for NOOP and KILL, which have the obvious effects.

We define the value of an action with three attributes, where only one of them is set to a non-zero value. For NOOP and KILL, we set the corresponding attributes to 1. For other actions, we use their cost, or time/space efficiency, as the value: *i.e.*, how much time the action needs per unit of space that it makes available. Then for an action a on a VM p, its value $value(p,a)$ is defined as:

$$\begin{cases} value(p,a).cost = \dfrac{time(p,a)}{space(p,a)}, & \text{for } \texttt{GROW}, \texttt{YGC}, \\ & \quad \texttt{FGC}_\texttt{p}, \texttt{FGC}_\texttt{c}, \\ value(p,a).\texttt{NOOP} = 1, & \text{for } \texttt{NOOP}, \\ value(p,a).\texttt{KILL} = 1, & \text{for } \texttt{KILL}, \end{cases}$$

With the above definition, our manager can favor actions by comparing these three attributes in a certain order, as we describe in Section 3.2.4.

3.2.4 Memory Allocation Algorithm

Next, we discuss the allocation algorithm, which allocates memory to the JVMs by performing actions on them at each timestep. We model the problem as a 0-1 knapsack problem. The capacity of the knapsack is the total amount of memory, and the items are actions performed on JVMs. Each action has a value and a minimum space requirement as described in Table 2. The goal is to maximize the total item value in the knapsack without exceeding its capacity.

The 0-1 knapsack problem is known to be NP-complete with a pseudo-polynomial dynamic programming solution [14]. Let $opt_{N,M}$ denote the value of the best scheme for allocating memory of size M to the first N JVMs, $p_1 \cdots p_N$. If a JVM p_i is undergoing a GC, the manager skips it to wait for the GC to complete. Otherwise, it derives $opt_{i,j}$ by enumerating possible actions on p_i and picking the one that leads to the largest value for $opt_{i,j}$. We define the sum of two values as the sum of their three attributes, then the state transition function is defined as:

$$opt_{i,j} = \begin{cases} opt_{i,j} & \text{if } opt_{i,j} > opt_{i-1,j-m} + v, \\ opt_{i-1,j-m} + v & \text{otherwise,} \end{cases}$$

where $v = value(p_i,a), a \in Actions, i \in [1,N], j \in [0,M]$.

To choose between two values, we first check which one has a lower value for attribute KILL, then fewer NOOPs to reduce pausing time, then a smaller time/space ratio. The one with fewer KILL, then fewer NOOP, then smaller time/space ratio, has a *higher* value. To be precise, given two values a and b, we define $a > b$ as:

```
bool operator>(const Value& a,
               const Value& b) {
  if a.KILL < b.KILL return true
  if a.NOOP < b.NOOP return true
  if a.cost < b.cost return true
  return false
}
```

The complexity of the dynamic programming is $O(N * M)$, where N is the number of JVMs and M is the total amount of memory. On modern servers, M can be large if the memory-size units are fine-grained, which would prevent the manager from making fast decisions. At the same time, allocating memory at fine granularity is unnecessary. To enable fast memory-allocation decisions, we define U as the unit of memory allocation, and any allocation is represented as a multiple of U. We discuss two ways of setting U: as a constant or as a dynamically computed variable based on the current heap state, and evaluate their impact on performance in Section 4.

Algorithm 1 and Algorithm 2 show the detailed allocation algorithms. Function ALLOCATE allocates memory of size M across the list of JVMs, P, at the current timestep, and it returns the best allocation scheme, act^{best}, which is a vector of actions for each $p \in P$. The algorithm works as follows: First, we find all the JVMs that are not undergoing a GC as $P - P_{\texttt{INGC}}$ to compute their actions. Because the algorithm allocates memory as increments of U, but $y_{limit}(p)$ and $o_{limit}(p)$ of a JVM p at the current timestep may not be increments of U when U is a dynamic variable, we do not include NOOP in Algorithm 2. Instead, we consider all the combinations of $P - P_{\texttt{INGC}}$ as potential $P_{\texttt{NOOP}}$ (line 4) and use $P' = P - (P_{\texttt{INGC}} \cup P_{\texttt{NOOP}})$ to denote the remaining JVMs. The remaining memory to be allocated is of size M' (line 7). We then apply Algorithm 2 on P' and memory of size M' ($= K$ units of size U). Function KNAPSACK returns the best solution with its value. The generation size limits and value of an action on a JVM are computed as in Table 2. The size limits are aligned to increments of U by function $align(size,U)$ defined as:

Action a	$\mathbf{y_{cap}}(\mathbf{p},\mathbf{a})$	$\mathbf{o_{cap}}(\mathbf{p},\mathbf{a})$	$\mathbf{space}(\mathbf{p},\mathbf{a})$	$\mathbf{time}(\mathbf{p},\mathbf{a})$
GROW	$y_{used}(p)+g\hat{r}w(p)$	$\hat{y}_{live}(p)+o_{used}(p)$	$y_{cap}(p,a)+o_{cap}(p,a)$ $-y_{limit}(p)-o_{limit}(p)$	$os(space(p,a))$
YGC	$y_{used}(p)$	$\hat{y}_{live}(p)+o_{used}(p)$	$\hat{y}_{dead}(p)$	$\hat{gc}_y(y_{obj}(p))$
$\texttt{FGC}_\texttt{p}$	$y_{used}(p)$	$\hat{y}_{live}(p)+o_{used}(p)$	$\hat{y}_{dead}(p)+\hat{o}_{dead}(p)$	$\hat{gc}_y(y_{obj}(p))+\hat{gc}_o(o_{obj}(p))$
$\texttt{FGC}_\texttt{c}$	$y_{used}(p)$	$o_{used}(p)$	$\hat{y}_{dead}(p)+\hat{o}_{dead}(p)$	$\hat{gc}_y(y_{obj}(p))+\hat{gc}_o(o_{obj}(p))*r,$ $r=(\hat{y}_{live}(p)+\hat{o}_{live}(p))/\hat{o}_{live}(p)$
NOOP	$y_{limit}(p)$	$o_{limit}(p)$		
KILL	0	0		

Table 2: Per-generation size limit requirements, sizes of created space, and time taken for each action a in $Actions$ on JVM p at the current timestep. $os(m)$ is the time to access memory of size m. Other symbols are defined in Table 1.

Algorithm 1 The scheduling algorithm: allocates memory of size M across the list of JVMs P, returns the allocation scheme.

```
1:  function ALLOCATE(P,M)
2:      value^best = act^best = None
3:      P_INGC = {p ∈ P, p is undergoing a GC}
4:      for P_NOOP ∈ power set of P − P_INGC do
5:          act_p = NOOP, p ∈ P_NOOP
6:          P' = P − (P_INGC ∪ P_NOOP)
7:          M' = M − Σ_{p∈P_INGC∪P_NOOP} (y_limit(p) + o_limit(p))
8:          Compute U, let K = M'/U
9:          act', value' = Knapsack(P', K, U)
10:         act_p = act'_p, p ∈ P'
11:         value.cost = value'.cost, value.KILL = value'.KILL
12:         value.NOOP = size of P_NOOP
13:         if value > value^best then
14:             value^best = value, act^best = act
15:     if act^best contains only NOOP then
16:         Pick P_kill ⊆ P, let act_p^best = KILL, p ∈ P_kill
17:     return act^best
```

Algorithm 2 The knapsack problem: given the list of JVMs P and K memory units of size U, returns the best allocation and its value.

```
1:  function KNAPSACK(P,K,U)
2:      N = size of P
3:      opt_{0,j} = 0, j ∈ [0,K]
4:      for i ← 1,N do
5:          for j ← 0,K do
6:              for a ∈ [GROW, YGC, FGC_c, FGC_p, KILL] do
7:                  if a ∈ [YGC, FGC_c, FGC_p] and
8:                      space(p_i,a) < mingcsave then continue
9:                  y_unit = align(y_cap(p_i,a),U)
10:                 o_unit = align(o_cap(p_i,a),U)
11:                 if opt_{i−1,j−y_unit−o_unit} is valid then
12:                     v = opt_{i−1,j−y_unit−o_unit} + value(p_i,a)
13:                     if v > opt_{i,j} then
14:                         opt_{i,j} = v, trans_{i,j} = (a, y_unit + o_unit)
15:     Derive act_p of each p ∈ P from opt_{N,K} and trans_{N,K}
16:     return act, opt_{N,K}
```

$align(size,U)=ceiling(size/U)$. For GC actions, we defined a constant *mingcsave* to avoid GCs that only recycle a negligible amount of space. We derive *act* from the transition actions *trans* and return them together with the value. They are then merged with $P_{\texttt{NOOP}}$ and $P_{\texttt{INGC}}$ to get the final allocation. We maintain the best allocation and its value across all the powersets. In the end, if the best allocation only contains NOOP actions, we pick some JVMs to kill to make progress. In this work, we pick the query that occupies the largest amount of memory and kill all its JVMs, and we leave other strategies as future work.

3.3 Estimating Runtime Values

The last piece of ElasticMem is the models that estimate JVM values that are necessary for memory allocation decisions yet not directly available as indicated in Table 1.

3.3.1 Heap Growth

To allocate memory to a JVM for the next timestep, the memory manager needs to estimate its memory growth. Different approaches are possible. In this paper, we adopt a simple approach. To estimate the heap growth of JVM p at timestep t, $g\hat{r}w(p,t)$, the manager maintains the maximum change in the young generation's usage during the past b timesteps. To be precise, we define: $g\hat{r}w(p,t)=\max|y_{used}(p,t')-y_{used}(p,t'-\delta_t)|, t'\in[t-b*\delta_t,t]$. In our experiments, we set $b=3$ empirically. We show in Section 4 that this value yields good performance.

3.3.2 GC Time and Space Saving

The GC time and space saving depend primarily on the number and total size of the live and dead objects in the collected region. Unfortunately, getting such detailed statistics is expensive, as we need to traverse the object reference graph similarly as in a GC. Paying such a cost for each JVM at every prediction defeats the purpose of reducing GC costs in the first place.

We observe, however, that a query operator's data structures and their update patterns determine the state of live and dead objects, which determines GC times and the amount of reclaimable memory. Our approach is thus to monitor the state of major data structures in query operators, collect statistics from them as features, and use these

Feature	Meaning
nt	Total # of processed tuples
ntd	Delta # of processed tuples since the last GC
nk	Total # of distinct keys in the hash table
nkd	Delta # of distinct keys since the last GC
num_{long}	# of `long` columns
num_{str}	# of `String` columns
sum_{str}	Avg. sum of lengths of all `String` columns

Table 3: Features collected from a hash table.

features to build models. While there are many operators in a big data system, most keep their state in a small set of data structures, for example, hash tables. So instead of changing the operators, we wrap data structures with the functionality to report statistics, and instrument them during query execution to get per-data structure statistics. There are many large data structures, but in data analytics systems, the most commonly used ones by operators with large in-memory state, such as join and aggregate, are hash tables. In this paper, we focus on the hash table data structure. To get predictions for the whole query, we first build models for one hash table, then compute the sum of per-hash-table predictions as the prediction for the whole query. Our approach, however, can easily be extended to other data structures and operators.

Table 3 lists the statistics that we collect for a hash table. A hash table stores tuples consist of columns. A tuple has a key defined by some columns and a value formed by the remaining columns. We collect the number of tuples and keys in a hash table in both generations (both the total and the delta since the previous GC), since new objects are put in the young generation only until a GC. These features are *nt*, *ntd*, *nk* and *nkd*. The schema also affects memory consumption. In particular, primitive types, such as `long`, are stored internally using primitive arrays (e.g. `long[]`) in many systems that optimize memory consumption. However, data structures with Java object types, such as `String`, cannot be handled in the same way, as their representations have large overhead. So we treat them separately by introducing features for primitive types (num_{long}) and `String` types (num_{str} and sum_{str}). The overhead of getting these values from hash tables is negligible. We then build machine learning models to predict the GC times and the total size of live and dead objects as specified in Table 1.

To build models, our first approach to collect training examples is to randomly trigger GCs during execution to collect statistics. The models built from them, however, yielded poor predictions for test points that happen to fall in regions with insufficient training data. As a second approach, we collected training data using a coarse-grained multidimensional grid with one dimension per feature. The examples were uniformly distributed throughout the feature space but they all had the same small set of distinct feature values, the values from the grid. As a result, predictions were excellent for values on the grid but poor otherwise. Using a fine-grained grid, however, is too expensive since the feature space has eight dimensions. For example, if we divide each dimension in four, the total number of grid points is $(4+1)^7 = 78,125$. Assuming that collecting one data point requires 30 seconds, we need $78,125/2/60 \approx 651$ machine hours. Our final approach is thus to combine the previous two: We first collect data using a coarse-grained grid to ensure uniform coverage of the entire feature space, then for each grid cell, we introduce some diversity by collecting two randomly selected data points inside of it. The union of the grid and the random points is the training set. To collect a data point for a hash table, we run a query with only that hash table and a synthetically generated dataset as the input. This approach enables us to precisely control the feature values when we trigger a GC. We then can use any off-the-shelf approach to build a regression model. In our implementation, we use the M5P model [40, 50] from Weka [20] since it gives us the most accurate predictions overall. We evaluate our models in Section 4.2.

4 Evaluation

We evaluate the performance of our memory manager and the accuracy of our models. We perform all experiments on Amazon EC2 using `r3.4xlarge` instances. We do not set swap space to avoid performance degradation due to virtual memory swapping. We execute TPC-H queries [6] on Myria [48], a shared-nothing data management and analytics system written in Java. The TPC-H queries are written in MyriaL, which is Myria's declarative query language, and they are publicly available at [7]. We modify or omit several queries because MyriaL does not support some language features, such as nulls and `ORDER BY`. The final set consists of 17 TPC-H queries: *Q1-Q6*, *Q8-Q12*, and *Q14-19*. To experiment with a broad range of query memory consumption, we execute each query on two databases with scale factors one and two.

4.1 Scheduling

We first compare our elastic manager (*Elastic*) against the original JVM with fixed maximum heap size (*Original*). For Original, we assume that each running JVM gets an equal share of the total memory. We pick 4 memory-intensive TPC-H queries, Q4, Q9, Q18, and Q19, and execute each on two databases, which leads to a total of 8 queries. In all experiments, we execute these 8 queries on one EC2 instance together with our memory manager. All data points are averages of five trials, and we report the minimal and maximal values as floating error bars. Each run of the allocation algorithm takes about 0.15 seconds.

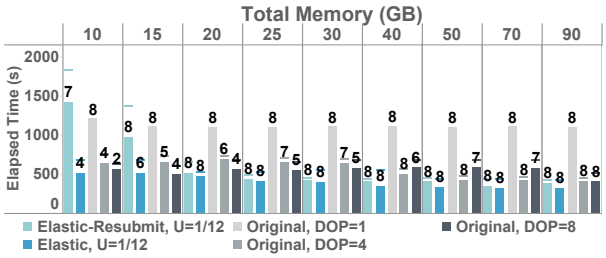


Figure 4: Average elapsed times and # of completed queries (labeled on top of each bar).

We empirically set the constant *mingcsave* from Algorithm 2 to 30 MB. The value of the function $os(m)$ is obtained by running a calibration program, which asks the operating system for memory of size m using `mmap` and accesses it using variable assignments. We take the system time as $os(m)$. For `r3.4xlarge`, we get $os(m) = 0.35s * \frac{m}{1\,\text{GB}}$. We set the interval between timesteps, δ_t, to 0.5 seconds except in Section 4.1.3, where we compare different values of δ_t. In order to avoid query hanging due to frequent GCs that do not recycle much memory, we kill a query after 8 minutes if it is still running. Based on our observation, 8 minutes is long enough for any query to complete with a reasonable amount of memory.

One extreme of Original is serial execution where queries are executed one at a time, while the other extreme is to execute all queries simultaneously. The former approach requires the least amount of memory for all queries to complete but takes longer time, while the latter finishes all queries the fastest when memory is sufficient, however may fail more queries when memory is scarce. We vary the degree of parallelism (*DOP*) for Original to compare these alternatives. To make it fair for Elastic, we also introduce a variant of Elastic, which allows executions to be delayed by resubmitting killed queries serially after all queries either complete or get killed. We call this variant *Elastic-Resubmit*. To avoid livelocks, we only resubmit each killed query once, and each resubmitted query runs only by itself. We leave resubmitting multiple queries simultaneously as future work.

Another important parameter is the size of the memory increment unit U. The value of U can be either fixed or derived in real time. We test fixed sizes of 100 MB, 500 MB, and 1000 MB, and variable sizes as 1/8, 1/12, and 1/16 of the total free space at the current timestep.

4.1.1 Scheduling Simultaneous Queries

First, we submit all queries at the same time. Figure 4 shows the elapsed times, together with the numbers of completed queries while varying the total memory size. The elapsed times are the times for all queries to complete. In this figure, we use U=1/12 as the representative of our elastic manager because it provides the best overall performance across all experiments. We further discuss the performance of different values of U in Figure 5. When memory is abundant ($\geq$ 20 GB), both Elastic managers yield more completed queries and also shorter elapsed times than all the three Original variants. When memory is scarce ($\leq$ 15 GB) and only suffices to execute one query at a time, for 15 GB, Elastic-Resubmit is able to complete all queries with less time than Original, DOP=1. For 10 GB, it only misses one query with a slightly longer time comparing to DOP=1. Based on our observation, the query failed because our manager needs to allocate memory as increments of U, however U is not sufficiently fine-grained. The overhead of elapsed time is due to the elastic method striving to accommodate all queries together before degrading to serial execution. As a proof of concept, we calculate the in-memory sizes of dominant large hash tables of the 8 queries and find that the sum of them is about 14 GB. This experiment shows the advantage of using the elastic manager: it automatically adjusts the degree of parallelism, enabling the system to get high-performance while avoiding out-of-memory failures when possible.

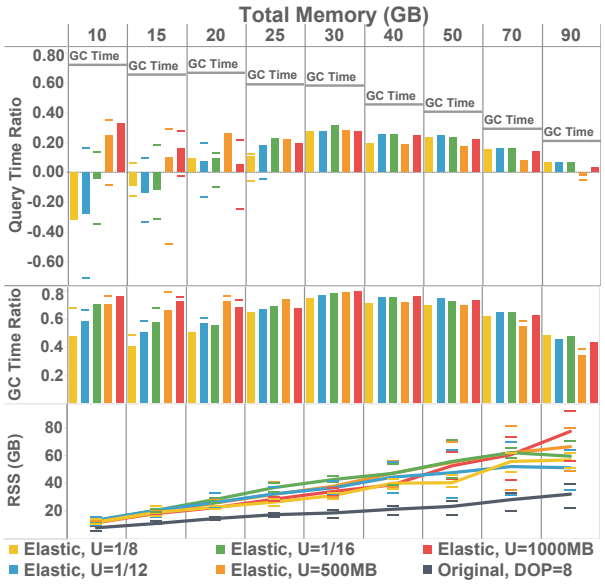


Figure 5: Relative total query time improvement ratio (top) and GC time improvement ratio (middle) for Elastic over Original, DOP=8, and resident set size, RSS (bottom).

In Figure 5, we further drill down on the performance of different variants of our approach. We seek to determine which variant yields the greatest performance improvement compared with non-elastic memory management. Because the elapsed times of Original, DOP=1 are significantly longer than the other two variants, we use Original, DOP=8 as the baseline in this experiment, which also brings fair comparison with our approach. We measure performance in terms of total query execution time, which is the sum of the per-query execution times, and

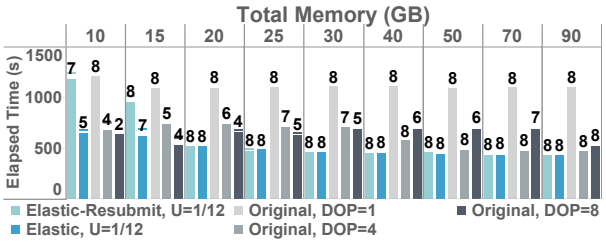


Figure 6: Average elapsed times and # of completed queries (labeled on top of each bar) with 30 seconds delay.

total GC time, the sum of the GC times of all queries. Figure 5 shows the relative improvement percentages in total query execution time and GC time of Elastic over Original, DOP=8, for different values of U, and also the actual physical memory usage (resident set size, RSS).[4] Higher bars indicate greater improvements. When memory is scarce ($\leq$ 15 GB), Elastic with variable values of U (1/8, 1/12 and 1/16) takes longer to execute each query because it strives to finish more queries than Original, DOP=8, as shown previously in Figure 4. When memory is abundant ($\geq$ 20 GB), for any of the values of U, Elastic outperforms Original, DOP=8 on both total query time and GC time. The percentage improvements are between 10% and 30% for query time and 40% to 80% for GC time. We observe that it is caused by Original, DOP=8 triggering GCs that do not recycle much space especially in late stages for large queries but being unable to shift memory quota from small queries, while Elastic can dynamically allocate memory across all queries. The improvement ratios of query time decrease after 70 GB because GC time takes a less portion of query time when memory is abundant. To show the maximum improvement that we can achieve by reducing GC time to zero, we also show the ratios of total GC time to query time in the top subfigure as a reference. Finally, the bottom subfigure shows that our elastic manager is also able to utilize a larger fraction of available physical memory to save on GC time and query time. Importantly, all values of U, especially the three variable ones, yield similar performance indicating that careful tuning is not required.

4.1.2 Scheduling Queries with Delays

To better simulate a real cluster, instead of issuing all the queries at the same time, we submit the above 8 queries with delays. Each query is submitted 30 seconds later than the previous one. Figure 6 shows the elapsed times and the numbers of completed queries. The patterns are similar to the experiment above with no delay (Figure 4), but also different as Elastic can finish the same number of queries with less time when memory is scarce (10 GB), and always beats all variants of Original in terms of both query completion and elapsed time. This is due to the memory flexibility that ElasticMem has: the number of simultaneously running queries is lower when delay is introduced, so Elastic is able to finish more queries faster, while Original stays the same.

4.1.3 Timestep Interval

Finally, we evaluate the sensitivity of the approach to different values of δ_t varying from 0.1, 0.5, or 1 second for U=500MB and U=1/12. We find that when memory is scarce, 0.5 seconds slightly outperforms others by completing more queries with less time, although in general the three δ_ts yield similar performance, which indicates that the approach is not sensitive to small differences when using variable sizes of U and thus careful tuning is not necessary. We omit details due to space constraints.

4.2 GC Models

An important component of ElasticMem is its models that predict the GC time and the space that will be freed (Section 3.3). We evaluate its models in this section. We limit the training space to 12 million tuples and 12 million keys for a hash table, with the schema varying from 1 to 7 `long` columns and 0 to 8 `String` columns with a total of 0 to 96 characters. This training space is large enough to fit all hash tables from TPC-H queries. As described in Section 3.3, we collect approximately 1080 grid points and 1082 random points together as the training set. We also collect a test set of 7696 data points by randomly triggering GC for the 17 TPC-H queries on both databases.

We set the JVM to use one thread for GC (`-XX:ParallelGCThreads=1`) because we observe that the JVM is not always able to distribute work evenly across multiple GC threads. We do not use thread-local buffers (`-XX:-UseTLAB`). We let the JVM always sweep live objects to the beginning of the old generation after each collection (`-XX:MarkSweepAlwaysCompactCount=1`) instead of every few collections to reduce GC cost variance. Among several models available in Weka [20], we pick the M5P model with default settings for its overall accuracy. M5P is a decision tree where leaves are linear regressions [40,50]. We use *relative absolute error* (RAE) to measure the prediction accuracies.[5]

Figure 7 shows the results for both doing 10-fold cross validation on the training set and testing on the random TPC-H test set. For cross validation, the predictions yield RAEs below 5% for every value except o_{dead}. For testing, both y_{dead} and o_{dead} cannot be predicted well, while all others have RAEs lower than 25%. This is because that

[4] We define the improvement percentage as $(x-y)/x$, where x is the value of Original and y is the value of Elastic.

[5] The RAE of a list of predictions P_i and corresponding real values R_i is defined as: $\sum_{i=1}^{n}|P_i-R_i|/\sum_{i=1}^{n}|\bar{R}-R_i|$.

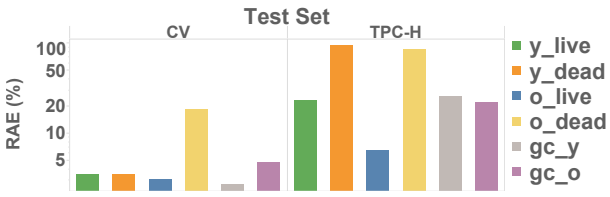


Figure 7: GC Model accuracies on 10-fold cross validation and random TPC-H test set.

the size of dead objects is not strongly correlated with the objects in data structures. Fortunately, the fact that the sum of dead and live objects is the total used size gives us a way to avoid predicting y_{dead} and o_{dead}. Instead, we let $\hat{y}_{dead} = y_{used} - \hat{y}_{live}$ and $\hat{o}_{dead} = o_{used} - \hat{o}_{live}$, where y_{used} and o_{used} can be obtained precisely. Overall, the prediction error rates are low and, as we showed in Section 4.1, suffice to achieve good memory allocation decisions.

5 Related Work

Memory allocation within a single machine: Many approaches focus on sharing memory across multiple objects on a single machine. Several techniques have *queries* as the objects: Some [12, 16, 38] allocate buffer space across queries based on page access models to reduce page faults. Others [11, 39] tune buffer allocation policies to meet performance goals in real-time database systems. A third set of methods [45] uses application resource sensitivities to guide allocation. More recently, Narasayya *et al.* [37] develop techniques to share a bufferpool across multiple *tenants*. Several approaches focus on *operators* within a query. Anciaux *et al.* [10] allocate memory across operators on memory-constrained devices. Davison *et al.* [15] sell resources to competing operators to maximize profit. Garofalakis *et al.* [17] schedule operators with multidimensional resource constraints in NUMA systems. Finally, Storm *et al.* [44] manage memory across *database system components*. Although they share the idea of managing memory for multiple objects with a global objective function, the problems are restricted to single machines, and they ignore GC. Salomie *et al.* [41] move memory across JVMs dynamically by adding a balloon space to OpenJDK but have no performance models or scheduling algorithms. Ginkgo [26] dynamically manages memory for multiple Java applications by changing layouts using Java Native Interface. However, it models performance by profiling specific workloads, while our approach is applicable to arbitrary relational queries.

Cluster-wide resource scheduling: Some techniques develop models to understand how resources affect the runtime characteristics of applications. Li *et al.* [32] partition queries on heterogeneous machines based on system calibrations and optimizer statistics. Herodotou *et al.* [23, 24] tune Hadoop application parameters based on machine learning models built by job profiles. Some other techniques focus on *short-lived requests*. Lang *et al.* [30] schedule transactional workloads on heterogeneous hardware resources for multiple tenants. Schaffner *et al.* [42] minimize tail latency of tenant response times in column database clusters. BlowFish [28] adaptively adjusts storage for performance of random access and search queries by switching between array layers with different sampling rates based on certain thresholds. In contrast, our focus is relational queries on Java-based systems with no sampling. To provide a unified framework for resource sharing and application scheduling, several *general-purpose* resource managers have emerged [25,47,51]. However, they all lack the ability to adjust memory limits dynamically.

Adaptive GC tuning: Cook *et al.* [13] provide two GC triggering policies based on real-time statistics, but do not investigate memory management across applications. Simo *et al.* [43] study the performance impact of JVM heap growth policies by evaluating them on several benchmarks. Maas *et al.* [35] observe that GC coordination is important for distributed applications. They let users specify coordination policy to make all JVMs trigger GC at the same time under certain conditions.

Region-based memory management: Another line of work uses region-based memory management (RBMM) [46] to avoid GC overhead. Broom [18] categorizes Naiad [36] objects into three types with a region assigned to each. Deca [34] manipulates Spark Scala objects in-memory representations as byte arrays and allocates pages for them. While RBMM may reduce GC overhead, it requires that the programmer declare object-to-region mappings and adds complexity to compilation, without eliminating space safety concerns [21].

6 Conclusion and Future Work

In this paper, we presented ElasticMem, an approach for the automatic and elastic memory management for big data analytics applications running in shared-nothing clusters. Our approach includes a technique to dynamically change JVM memory limits, an approach to model memory usage and garbage collection cost during query execution, and a memory manager that performs actions on JVMs to reduce total failures and run times. We evaluated our approach in Myria and showed that our approach outperformed static memory allocation both on query failures and execution times. We leave extensions to other data structures and experiments with more diverse workloads and systems as future work.

Acknowledgment: This project is supported in part by the National Science Foundation through grant IIS-1247469 and the Intel Science and Technology Center for Big Data.

References

[1] Apache Flink. http://flink.apache.org/.

[2] Apache Giraph. http://giraph.apache.org/.

[3] Docker Container. https://www.docker.com/.

[4] ElasticMem. http://myria.cs.washington.edu/projects/2015/09/12/cloud_service.html#elasticmem.

[5] Kubernetes. http://kubernetes.io/.

[6] TPC-H. http://www.tpc.org/tpch/.

[7] TPC-H queries in MyriaL. https://github.com/uwescience/tpch-myrial.

[8] Tungsten: memory management and binary processing on Spark. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

[9] Memory management in the Java HotSpot™ virtual machine. http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf, 2006.

[10] N. Anciaux, L. Bouganim, and P. Pucheral. Memory requirements for query execution in highly constrained devices. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 694–705. VLDB Endowment, 2003.

[11] K. P. Brown, M. J. Carey, and M. Livny. Managing memory to meet multiclass workload response time goals. In *Proceedings of the 19th International Conference on Very Large Data Bases*, VLDB '93, pages 328–341, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[12] C.-M. Chen and N. Roussopoulos. Adaptive database buffer allocation using query feedback. In *Proceedings of the 19th International Conference on Very Large Data Bases*, VLDB '93, pages 342–353, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[13] J. E. Cook, A. W. Klauser, A. L. Wolf, and B. G. Zorn. Semi-automatic, self-adaptive control of garbage collection rates in object databases. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 377–388, New York, NY, USA, 1996. ACM.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, chapter 16, pages 425–427. The MIT Press, 3nd edition, 2009.

[15] D. L. Davison and G. Graefe. Dynamic resource brokering for multi-user query execution. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 281–292, New York, NY, USA, 1995. ACM.

[16] C. Faloutsos, R. T. Ng, and T. K. Sellis. Predictive load control for flexible buffer allocation. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, pages 265–274, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[17] M. N. Garofalakis and Y. E. Ioannidis. Parallel query scheduling and optimization with time- and space-shared resources. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 296–305, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[18] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingam, M. Costa, D. G. Murray, S. Hand, and M. Isard. Broom: Sweeping out garbage collection from big data systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, 2015. USENIX Association.

[19] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993.

[20] 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. 2009.

[21] N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 141–152, New York, NY, USA, 2002. ACM.

[22] D. Halperin, V. Teixeira de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker, S. Xu, M. Balazinska, B. Howe, and D. Suciu. Demonstration of the Myria big data management service. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 881–884, New York, NY, USA, 2014. ACM.

[23] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 18:1–18:14, New York, NY, USA, 2011. ACM.

[24] 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 *In CIDR*, pages 261–272, 2011.

[25] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. 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*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[26] M. R. Hines, A. Gordon, M. Silva, D. Da Silva, K. Ryu, and M. Ben-Yehuda. Applications know best: Performance-driven memory overcommit with ginkgo. In *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, CLOUDCOM '11, pages 130–137, Washington, DC, USA, 2011. IEEE Computer Society.

[27] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, SIGMOD '91, pages 268–277, New York, NY, USA, 1991. ACM.

[28] A. Khandelwal, R. Agarwal, and I. Stoica. Blowfish: Dynamic storage-performance tradeoff in data stores. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 485–500, Berkeley, CA, USA, 2016. USENIX Association.

[29] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR*, 2015.

[30] W. Lang, S. Shankar, J. M. Patel, and A. Kalhan. Towards multi-tenant performance slos. *IEEE Trans. on Knowl. and Data Eng.*, 26(6):1447–1463, June 2014.

[31] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, Nov. 2015.

[32] J. Li, J. Naughton, and R. V. Nehme. Resource bricolage for parallel database systems. *Proc. VLDB Endow.*, 8(1):25–36, Sept. 2014.

[33] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[34] L. Lu, X. Shi, Y. Zhou, X. Zhang, H. Jin, C. Pei, L. He, and Y. Geng. Lifetime-based memory management for distributed data processing systems. *Proc. VLDB Endow.*, 9(12):936–947, Aug. 2016.

[35] M. Maas, T. Harris, K. Asanovic, and J. Kubiatowicz. Trash day: Coordinating garbage collection in distributed systems. In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*, 2015.

[36] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[37] V. Narasayya, I. Menache, M. Singh, F. Li, M. Syamala, and S. Chaudhuri. Sharing buffer pool memory in multi-tenant relational database-as-a-service. *Proc. VLDB Endow.*, 8(7):726–737, Feb. 2015.

[38] R. Ng, C. Faloutsos, and T. Sellis. Flexible buffer allocation based on marginal gains. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, SIGMOD '91, pages 387–396, New York, NY, USA, 1991. ACM.

[39] H. H. Pang, M. J. Carey, and M. Livny. Managing memory for real-time queries. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD '94, pages 221–232, New York, NY, USA, 1994. ACM.

[40] R. J. Quinlan. Learning with continuous classes. In *5th Australian Joint Conference on Artificial Intelligence*, pages 343–348, Singapore, 1992. World Scientific.

[41] T.-I. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone. Application level ballooning for efficient server consolidation. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 337–350, New York, NY, USA, 2013. ACM.

[42] J. Schaffner, B. Eckart, D. Jacobs, C. Schwarz, H. Plattner, and A. Zeier. Predicting in-memory database performance for automating cluster management tasks. In *ICDE*, pages 1264–1275. IEEE Computer Society, 2011.

[43] J. Simão and L. Veiga. Adaptability driven by quality of execution in high level virtual machines for shared cloud environments. *Comput. Syst. Sci. Eng.*, 28(6), 2013.

[44] A. J. Storm, C. Garcia-Arellano, S. S. Lightstone, Y. Diao, and M. Surendra. Adaptive self-tuning memory in db2. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 1081–1092. VLDB Endowment, 2006.

[45] P. Tembey, A. Gavrilovska, and K. Schwan. Merlin: Application- and platform-aware resource allocation in consolidated server systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 14:1–14:14, New York, NY, USA, 2014. ACM.

[46] M. Tofte and J.-P. Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, Feb. 1997.

[47] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[48] 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.

[49] J. Wang and M. Balazinska. Toward elastic memory management for cloud data analytics. In *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, BeyondMR '16, pages 7:1–7:4, New York, NY, USA, 2016. ACM.

[50] Y. Wang and I. H. Witten. Induction of model trees for predicting continuous classes. In *Poster papers of the 9th European Conference on Machine Learning*. Springer, 1997.

[51] M. Weimer, Y. Chen, B.-G. Chun, T. Condie, C. Curino, C. Douglas, Y. Lee, T. Majestro, D. Malkhi, S. Matusevych, B. Myers, S. Narayanamurthy, R. Ramakrishnan, S. Rao, R. Sears, B. Sezgin, and J. Wang. Reef: Retainable evaluator execution framework. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1343–1355, New York, NY, USA, 2015. ACM.

[52] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[53] 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*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

Improving File System Performance of Mobile Storage Systems Using a Decoupled Defragmenter

Sangwook Shane Hahn, Sungjin Lee[†], Cheng Ji[*], Li-Pin Chang[‡],
Inhyuk Yee, Liang Shi[§], Chun Jason Xue[*], and Jihong Kim
Seoul National University, [†]Daegu Gyeongbuk Institute of Science and Technology,
[*]City University of Hong Kong, [‡]National Chiao-Tung University, [§]Chongqing University

Abstract

In this paper, we comprehensively investigate the file fragmentation problem on mobile flash storage. From our evaluation study with real Android smartphones, we observed two interesting points on file fragmentation on flash storage. First, defragmentation on mobile flash storage is essential for high I/O performance on Android smartphones because file fragmentation, which is a recurring problem (even after defragmentation), can significantly degrade I/O performance. Second, file fragmentation affects flash storage quite differently than HDDs. When files are fragmented on flash storage, the logical fragmentation and the physical fragmentation are decoupled and a performance degradation mostly comes from logical fragmentation. Motivated by our observations, we propose a novel defragger, janus defragger (janusd), which supports two defraggers, janusdL for a logical defragger and janusdP for a physical defragger. JanusdL, which takes advantage of flash storage's internal logical to physical mapping table, supports logical defragmentation without data copies. JanusdL is very effective for most fragmented files while not sacrificing the flash lifetime. JanusdP, which is useful for physically fragmented files but requires data copies, is invoked only when absolutely necessary. By adaptively selecting janusdL and janusdP, janusd achieves the effect of full file defragmentation without reducing the flash lifetime. Our experimental results show that janusd can achieve at least the same level of I/O performance improvement as e4defrag without affecting the flash lifetime, thus making janusd an attractive defragmentation solution for mobile flash storage.

1 Introduction

When a file system becomes highly fragmented, it has to allocate multiple split storage areas, i.e., extents [1], for a single file more frequently. In an HDD-based file system, accessing such a highly-fragmented file degrades the performance significantly due to the increased time-consuming seek operations. In order to mitigate the performance impact caused by file fragmentation, many file systems recommends the periodical execution of the defragmentation utility (e.g., every week) [2-6].

Unlike for HDD-based file systems, defragmentation is generally not recommended for flash-based file systems [7-13]. Since flash storage does not require seek operations, it is believed that the effect of defragmentation on the file system performance is rather negligible for flash storage. Furthermore, since a large number of files need to be copied during defragmentation, frequent defragmentation can affect the limited lifetime. However, this negative view toward flash defragmentation has been widely accepted without a proper validation study. The main goal of this paper, therefore, is to investigate the file fragmentation problem on mobile flash storage in a systematic and comprehensive fashion. Fig. 1 summarizes the key steps of our investigation study.

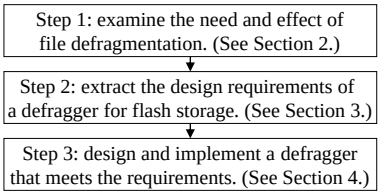


Fig. 1: A summary of the key steps in our investigation.

Since previous studies (e.g., [22]) have shown that files can be severely fragmented on mobile flash storage, in our study, we start with two key questions related to the effect of file defragmentation (step 1 in Fig. 1): 1) when fragmented files are defragmented, how much I/O performance is improved? and 2) how long does the effect of file defragmentation last? Unlike a common misconception on flash defragmentation, our evaluation study showed that I/O performance of flash storage can be significantly improved by defragmentation. For example, when fragmented files were defragmented, the average app launching time, which is an important user-perceived performance metric on smartphones, can be improved by up to 52% over the fragmented files.

Although fragmented files can degrade the I/O performance, if the effect of file defragmentation can last for long time (e.g., several months), a conventional defragmentation tool will be sufficient. However, our evaluation study indicated that file fragmentation may recur in a short cycle, around a week, even after full file defragmentation on smartphones. One main cause of recurring file fragmentation was frequent automatic app updates on smartphones. Since many popular apps tend to be updated very frequently (e.g., every 10 days [28]), the effect of file defragmentation quickly disappears.

When file defragmentation is repeatedly required, a conventional defragger such as e4defrag may not be an

appropriate solution for flash storage because it requires a large amount of data copies during defragmentation, thus seriously affecting the flash lifetime. For example, if we invoke e4defrag every week as suggested from our evaluation study, it might reduce the flash lifetime by more than 10%. Therefore, in order to maintain high I/O performance in a sustainable fashion, we need a different approach to the defragmentation problem for mobile flash storage, so that the impact of file defragmentation on the flash lifetime is less adverse.

The key insight behind janus defragger (janusd) comes from our investigation on the characteristics of file fragmentation in flash storage (step 2 in Fig. 1). Our study showed that file fragmentation affects flash storage quite differently from HDDs. In HDDs, when a (logical) file is highly fragmented, its physical layout is fragmented similarly with many isolated physical fragments. That means, logical fragmentation at the file system and physical fragmentation at the storage medium level are highly *correlated*. On the other hand, in flash storage, there is no physical counterpart at the storage medium level which is strongly correlated with logical fragmentation at the file system. For example, unlike HDDs where a degree of logical fragmentation directly affects the I/O performance at the storage medium level, the I/O performance at the storage medium level in flash storage is largely decided by an average degree of the I/O parallelism during I/O operations [16-21]. As will be explained in Section 3, since the average degree of the I/O parallelism for accessing a file is not correlated with the degree of logical fragmentation of the file, file fragmentation in flash storage occurs in a *decoupled* fashion between the logical space and the physical space. (In this paper, we call that a file `foo` is *physically fragmented* when the degree of the I/O parallelism in accessing `foo` is limited.)

In order to understand the impact of decoupled fragmentation on I/O performance, we evaluated the performance impact of file fragmentation on the entire mobile I/O stack layers. As expected, because of a high degree of the I/O parallelism at the storage medium level, only a small number of (unlucky) files were stored in a severely skewed fashion, limiting their I/O parallelism levels significantly. That is, regardless of how files were logically fragmented, their I/O performance at the storage medium level did not change much. On the other hand, logically fragmented files significantly increased processing times in the block I/O layer and the device driver because of a large increase in the number of block I/O requests. Therefore, the minimum requirement for a flash defragger would be to defragment the logical space effectively. Furthermore, since flash files are fragmented in a decoupled fashion, an ideal flash defragger needs to support an *independent* physical defragger as well. The physical defragger is necessary because a logical defragger cannot even identify physically fragmented files.

Motivated by the above requirements on a defragger for mobile flash storage, we propose a novel decoupled defragger, janusd, which consists of two defraggers, janusdL for a logical defragger and janusdP for a physical defragger (step 3 in Fig. 1). JanusdL, which takes advantage of flash storage's internal logical to physical mapping table, supports logical defragmentation without reducing the flash lifetime by avoiding explicit data copies. JanusdP, which independently operates from janusdL, works like a conventional defragger with data copies. Since the I/O performance of flash storage is dominated by logical file fragmentation, janusdL works very well for most fragmented files without affecting the flash lifetime. On a rare occasion when a file is physically fragmented, janusdP is invoked to restore the degraded file performance.

In order to validate the effectiveness of the proposed janusd technique, we have implemented janusd on an emulated mobile flash storage, `simeMMC` and `simUFS`. (`SimeMMC` and `simUFS`, which are based on an extended Samsung 843T SSD which supports host-level FTLs, are configured to effectively simulate the bandwidth of eMMC and UFS devices [14, 15], respectively.) Our experimental results show that janusd significantly improves the I/O performance of mobile flash storage. For example, janusd can reduce the app launching time by up to 53%, achieving an equivalent I/O performance improvement as e4defrag. However, janusd requires a less than 1% of data copies over e4defrag, thus making it an attractive defragmentation solution for flash storage. Furthermore, janusdL alone achieves about 97% of the janusd's performance level for most files.

The remainder of this paper is organized as follows. In Section 2, we report our key findings through our evaluation study of real-world file fragmentation on Android smartphones. Section 3 describes decoupled fragmentation in flash storage and explains needs for both logical and physical defraggers. A detailed description of janusd 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 File Fragmentation: User Study

In this section, we empirically investigate how file I/O performance is affected by file fragmentation on flash storage using 14 smartphones in use. In particular, we examine how quickly file fragmentation occurs again after defragmentation and how much I/O performance is affected by different defragmentation intervals.

2.1 Evaluation Study Setup

For our study, we collected 14 used Android smartphones. In order to avoid possible bias, we have se-

Table 1: File system utilizations of 14 smartphones.

50-59%	60-69%	70-79%	80-89%	90-99%
S5, GP	**S3**, G5	**N5**	**N6**, T2, T5, Z1, Z3	**S6**, I2, T3, T4

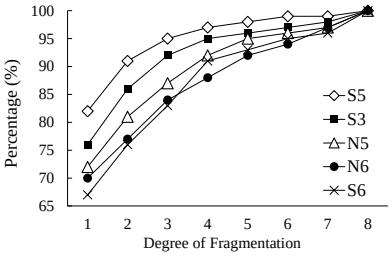


Fig. 2: Cumulative distributions of DoF values.

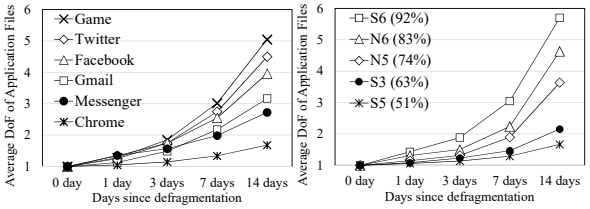
lected these smartphones from five different manufacturers with at least six month's real use. 14 users, like most other smartphone users, heavily used popular Android applications such as `Chrome`, `Messenger`, `Gmail`, `Facebook`, `Twitter` and `Game`. Table 1 divides 14 smartphones[1] into 5 categories based on the file system utilization. (In the rest of this section, we report the evaluation results on five representative smartphones, S5, S3, N5, N6 and S6, which were chosen from each utilization category.) We inspected file fragmentation on the `data` partition only because the `data` partitions occupied most of the total storage space available and most I/O operations occur in the `data` partition.

For our study, we used the degree $DoF(x)$ of fragmentation of a file `x`, which is defined as the ratio of the number of extents allocated to the file `x` to the ideal (i.e., smallest) number of extents needed for the file `x`. For example, if an 1-GB file `foo` in Ext4 were allocated to 24 extents, DoF(`foo`) would be 3 (i.e., 24/8), because `foo` would have required at least 8 extents even when `foo` was contiguously allocated. (A single extent can cover up to 128 MB in Ext4.) The large DoF value means that the file is highly fragmented.

2.2 Degree of File Fragmentation Analysis

We first examined DoF values of files in the `data` partition of the five smartphones using `e4defrag`, and Fig. 2 shows cumulative distributions of DoF values on the five smartphones. As reported in other investigations such as [22], our inspected smartphones exhibited similar characteristics on file fragmentation. Fragmented files accounted for between 14% and 33% of all files. In particular, on N5, 717 files among its 2,704 files were fragmented. Furthermore, 476 files were fragmented with their DoF values larger than 2. When the file system space was highly utilized, the number of fragmented files tends to be large. For example, on S6, having the highest file system utilization, 33% of its files were fragmented.

[1] 14 phones include Nexus 5 (N5), 6 (N6), Galaxy S3 (S3), S5 (S5), S6 (S6), Note 2 (T2), Note 3 (T3), Note 4 (T4), Note 5 (T5), Xperia Z1 (Z1), Z3 (Z3), Optimus G Pro (GP), G5 (G5) and Vega Iron 2 (I2).



(a) Six applications on N6. (b) `Twitter` on five smartphones.

Fig. 3: The average DoF value of application files.

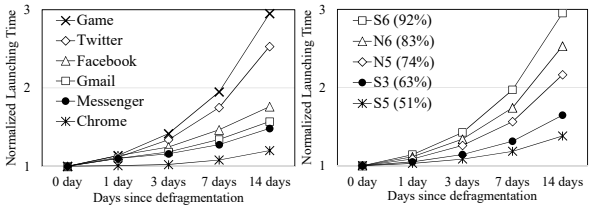
2.3 File Fragmentation Recurrence

Since our target smartphones have never been defragmented before, the results shown in Fig. 2 are interesting but somewhat expected. A more critical question for our study was to find out how soon file fragmentation recurs after full file defragmentation. If the recurrence interval of file fragmentation were quite large (say, several months), an existing defragmentation would be sufficient for mobile flash storage as well.

In order to understand file fragmentation recurrence (as well as others), after defragmenting all the files using `e4defrag`, we collected a daily snapshot of each smartphone for the subsequent two-week interval using a custom data collection app. Our snapshot data include DoF values of files and app launching times, Fig. 3(a) shows the changes in the average DoF values of the files associated with six popular applications, `Chrome`, `Messenger`, `Gmail`, `Facebook`, `Twitter` and `Game`, on N6. As shown in Fig. 3(a), file fragmentation recurred quickly after the full file system defragmentation. For most applications on N6, file fragmentation occurs again in a week since the full defragmentation. Fig. 3(b) shows the changes in the average DoF values of the files associated with `Twitter` on the five smartphones with different file system utilizations. The recurrence interval of file fragmentation was proportional to the file system utilization. For example, on the seventh day after the full file system defragmentation, the average DoF value of the `Twitter` files reached 1.86 and 3.04 for 70% and 90% of file system utilization, respectively. Even though only the DoF values of `Twitter` files are presented here, we had similar observations on the files of the other applications [42].

Our observation strongly suggests that file fragmentation is a recurring problem in smartphones, especially when the file system utilization is high[2]. In the following subsections, we shall show that file fragmentation negatively impact on user experience, but regular file defragmentation is harmful to flash storage lifetime. The proposed `janusd` technique is novel in that these two conflicting phenomena are resolved in a satisfactory fashion.

[2] One of the reasons for a short recurrence interval is frequent app updates which automatically invoked in background when a smartphone is connected to a Wi-Fi environment. Since popular apps such as `Twitter` are reported to be updated, on average, every 7 days [29], when the file system utilization is high, newly installed apps are very likely to experience severe file fragmentation.



(a) Six applications on N6. (b) `Twitter` on five smartphones.

Fig. 4: Changes in app launching times.

2.4 Impact on User Experience

File fragmentation can negatively impact on the smartphone user experience due to degraded I/O performance. For example, the launching of an application involves reading a set of files, including executables, libraries, and data files. This procedure creates a user-perceived latency because the user has to wait until all the required files have been loaded from flash storage. We define the launching time of an application to be the time interval between the time when the application icon is touched and the time when all graphical user interface components are displayed for the next user interaction.

Fig. 4(a) shows the launching time of the six popular applications on N6 and Fig. 4(b) depicts the launching time of `Twitter` on five smartphones with different file system utilizations. The launching time noticeably degraded as the day count increased, especially with the high file system utilization. For example, compared to the launching time right after the full file system defragmentation, the launching time of `Twitter` on the seventh day was already 1.6 times longer when the file system utilization was 70%, and the launching time was amplified to two times longer when the file system utilization was 90%. This result indicates that the recurring file fragmentation can highly impact the quality of user experience in a short period of time.

2.5 Impact on Flash Memory Lifetime

Because file fragmentation is a recurring problem, regular file defragmentation might be necessary to maintain satisfiable user experience. In fact, weekly file defragmentation is recommended by many defragmentation tools [25, 26]. However, conventional file defragmentation is based on data copies, which increases the wear in flash memory. We performed full file system defragmentation with different frequencies, including a daily basis and a weekly basis, under the emulated application update behaviors. Fig. 5 shows the total write traffic contributed by file defragmentation measured by the built-in Linux block I/O tracing tool `blktrace`. Surprisingly, the amount of data copies during file defragmentation was fairly large. For example, defragmenting files on the third day involved 1.8 GB of data copies under a 70% file system utilization, and this number increased to 5.76 GB if the file system utilization was 90%. If file defragmentation was performed in a weekly manner, the amount of data copies reached up to 9.53 GB.

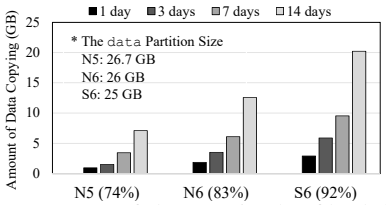


Fig. 5: The amount of data copies by file defragmentation with different defragmentation periods.

The extra data copies negatively impacts on flash memory lifetime. This problem is further exaggerated by the deteriorated flash endurance due to the introduction of multilevel cells. Specifically, the program-erase cycle (PE cycle) limit of TLC NAND is as low as 300 PE cycles. The `data` partition of the S6 is 25 GB, and weekly file defragmentation costs every flash block (9.53 GB/week $\times$ 4 weeks)/25 GB$\approx$1.5 extra PE cycles per month. In the typical smartphone life cycle of two years, weekly file defragmentation introduces 36 extra PE cycles to every block, and thus the flash lifetime is degraded by more than 10%. This significant lifetime reduction highly discourages the use of conventional copy-based file defragmentation tools on flash storage.

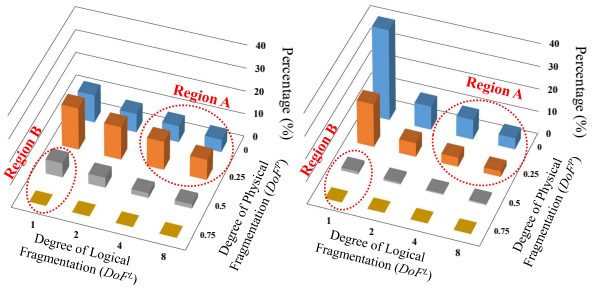
3 File Fragmentation: Under The Hood

In order to develop a flash-aware file defragmentation tool which does not have a negative effect on the flash lifetime, we performed a detailed characterization study of file fragmentation on flash storage.

3.1 Decoupled Fragmentation on Flash

Since flash storage works quite differently from HDDs at the storage medium level, before our study, we redefined the concept of physical fragmentation for flash storage.

Since flash storage is composed of a group of parallel I/O units (e.g., multiple flash memory channels/planes) and each I/O unit can support random access, a conventional definition of physical data sequentiality on hard drives does not make much sense to flash storage. In order to better reflect the effect of file fragmentation on I/O performance in flash storage, we associate two metrics, $DoF^L(\mathtt{x})$ and $DoF^P(\mathtt{x})$, for a file `x`, where $DoF^L(\mathtt{x})$ and $DoF^P(\mathtt{x})$ represent the degrees of logical fragmentation and physical fragmentation, respectively. For the logical DoF value, $DoF^L(\mathtt{x})$, of a file `x`, we use $DoF(\mathtt{x})$ as defined in Section 2.1. Since the I/O performance at the flash device level is largely determined by a degree of the I/O parallelism while accessing the file `x`, not the number of split extents as in HDDs, we define the physical DoF value, $DoF^P(\mathtt{x})$, of a file `x` as (1 - $DoP(\mathtt{x})$). $DoP(\mathtt{x})$, which indicates the effective degree of the I/O parallelism for accessing the file `x`, is computed as the ratio of the average degree of the I/O parallelism for accessing the file `x` sequentially to the maximum degree of the I/O parallelism supported by a flash storage sys-



(a) After 1-year aging. (b) After 1-week aging.

Fig. 6: A snapshot distribution of files classified based on their DoF^L values and DoF^P values.

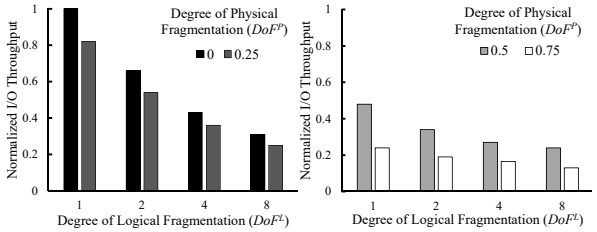
tem. When a flash storage system can support up to M I/O operations at the same time, if, on average, n operations were supported in parallel while accessing `foo`, DoP(`foo`) is n/M. Therefore, DoF^P(`x`) becomes 0 when the file `x` was accessed under the maximum I/O parallelism. As the effective degree of the I/O parallelism drops, DoF^P(`x`) approaches (1- 1/M).

In order to understand how logical fragmentation and physical fragmentation interact with each other in flash storage, we measured how DoF^L and DoF^P values change from the Ext4 file system after aging Ext4 with simulated one-year and one-week workloads. Since we need to collect DoF^P values, we used a mobile flash storage emulator (see Section 5).

Fig. 6 shows the distributions of DoF^L and DoF^P values after aging Ext4 with simulated one-year and one-week workloads, respectively. The results indicate that logical and physical fragmentation are highly decoupled. For example, the files in Region A suffered from high degrees of logical fragmentation but their degrees of physical fragmentation were quite low. On the other hand, surprisingly, there were still a few files in Region B that were barely fragmented at the logical space but suffered from high degrees of physical fragmentation.

Decoupled logical and physical fragmentation is mainly attributed to the high degree of the I/O parallelism available in flash storage as well as the extra indirection layer in flash storage for logical to physical mapping. Logical fragmentation and physical fragmentation impose different impacts on I/O performance. Specifically, logical fragmentation amplifies the overhead in the system software I/O stack due to the increased I/O frequency, while physical fragmentation degrades the I/O parallelism in flash storage. Defragmentation only at the logical or physical level may not produce the optimal I/O performance. For example, even though a file has been defragmented at the file system level, it dost not guarantee that the file is accessed through the maximum I/O parallelism inside of flash storage.

Conventional defragmentation tools cannot perform physical defragmentation for flash storage because the host does not have direct access to flash channels. In



(a) Varying DoF^L under low DoF^P's. (b) Varying DoF^L under high DoF^P's.

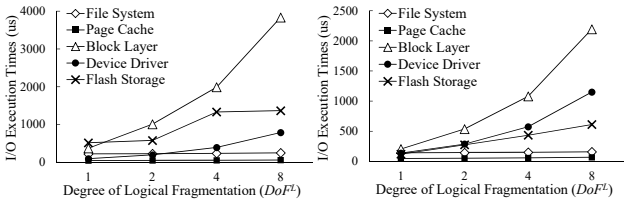
Fig. 7: I/O throughput under varying DoF^L and DoF^P.

addition, these tools are not aware of the existing indirection layer inside of flash storage, which is useful to modify the logical layout of files without physical data copies. We believe that the firmware of flash storage must be adequately involved during the defragmentation process. As shown in Fig. 6, the majority of file fragmentation is affiliated with logical fragmentation. While it is possible to perform copyless defragmentation for logically fragmented files (the files in Region A), data copies are still necessary to re-distribute data among flash channels for physical defragmentation. Fortunately, as shown in Fig. 6(a) and 6(b), the files with $DoF^P \geq 0.5$ contribute to no more than 20% of all files. In other words, physical defragmentation will be performed only for absolutely needed cases to prevent the extra data copies which will reduce the flash memory lifetime.

3.2 Need for Logical Defragmentation

To measure the significance of logical and physical fragmentation in terms of performance impact, we measured the throughput of reading a file `foo` under different values of DoF^L(`foo`) and DoF^P(`foo`). In order to control DoF^L values in our study, we made a simple utility which repeatedly splits a given file `foo` until DoF^L(`foo`) reaches the target DoF^L number. The performance measurement was conducted on the mobile flash storage emulator so that the degree of physical fragmentation DoF^P(`foo`) can also be carefully controlled. Based on the majority of the distribution in Fig. 6, the DoF^L value was between 1 and 8, while the DoF^P value was between 0 and 0.25. Fig. 7(a) shows that, when there was no physical fragmentation (DoF^P = 0), a high degree of logical fragmentation (DoF^L = 8) significantly degraded the I/O throughput by 75% compared to the case without any logical fragmentation (DoF^L = 1). On the other hand, increasing DoF^P(`foo`) from 0 to 0.25 only slightly degraded the throughput, no more than 20% for each DoF^L value. This observation suggests that logical fragmentation should be managed in a more aggressive manner than physical fragmentation.

In order to understand how logical fragmentation affects the overhead in the system software I/O stack, we built a fully integrated storage I/O profiler, IOPro, for quantitative evaluations. IOPro can profile the complete Android I/O stack from the application level to the de-



(a) Execution time changes on N6 (b) Execution time changes on S6

Fig. 8: Execution time changes under varying DoF^L.

vice driver level. The key feature of IOPro is that all I/O activities can be seamlessly linked together via their corresponding file information throughout the entire Android I/O stack. Using this tool, we can easily measure times spent in different I/O stack layers. For each measurement run, IOPro measured execution times spent in the Ext4 file system, the page cache, the block layer, the device driver and the mobile flash storage , respectively, on each of our inspected smartphones. For brevity's sake, we only include the measurement data for N6 and S6 in this section, which represent smartphones with eMMC devices and with UFS devices, respectively.

In order to evaluate the effect of logical fragmentation, we measured I/O execution times while varying DoF^L from 1 (no fragmentation) to 8 (heavy fragmentation). For all the measurements, we ran a simple synthetic I/O workload which reads a 512-KB file. The 512-KB file was pre-split into multiple fragments by our fragmentation utility so that the target DoF^L can be satisfied. Figs. 8(a) and 8(b) show how different I/O stack layers were affected under varying DoF^L values on N6 and S6, respectively. The times spent for the block layer, the device driver, and the flash storage device have increased as with the increasing DoF^L values. On the other hand, the times spent in the file system and page cache layers are barely affected. (In the block layer and the device driver, the increased number of block I/O requests in accessing the fragmented file directly affected the overhead of the I/O scheduler, handshaking and interrupt handling [36-41].) In mobile flash storage, although the same I/O layers were affected as in HDDs by the increased number of block I/O requests, the relative impact on these I/O layers were quite different from that in HDDs. As shown in Figs. 8, the block layer is dominantly affected by the number of block I/O requests over the flash storage device. In HDDs, the impact on the HDD device would have been very dominant, making the impact on the rest of I/O layers negligible.

3.3 Need for Physical Defragmentation

As previously shown in Fig. 6, most of the files have small DoF^P values (≤ 0.25). This is because, with the rich I/O parallelism inside of flash storage, it is very unlikely that a file suffers from extremely low I/O parallelism. For example, suppose that data are allocated among eight channels of equal availability, the probability that a 64-KB file composed of eight 8-KB flash pages is entirely allocated to one single channel would be 0.00004%. This probability further reduces if the file size is larger than 64 KB. On the other hand, the probability that the 64-KB file is allocated to 6 or more channels would around 80%.

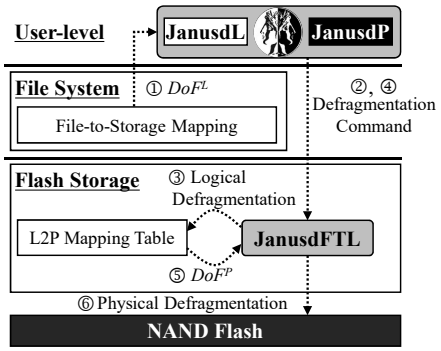


Fig. 9: An overall architecture of janusd.

Although it is a rare case that a file has a very high DoF^P value, the overall performance may still be adversely affected if a physically fragmented file is frequently accessed. Fig. 7(b) shows that, a high degree of physical fragmentation (i.e., ≥ 0.5) severely degraded the I/O throughput even when the degree of logical fragmentation was low. For example, even if a file was not fragmented at all in the logical space (DoF^L=1), if the file had a DoF^P value of 0.5, the I/O throughput became only 48% of that with DoF^P=0. Because logical and physical fragmentation is decoupled on flash storage, in such a rare case of high physical fragmentation, it is not sufficient to perform logical defragmentation only, and physical defragmentation is necessary to redistribute data among channels at a cost of flash lifetime.

4 Design and Implementation of Janusd

Our analysis in Sections 2 and 3 strongly indicates that file system fragmentation causes serious performance degradation even in flash storage, badly affecting the quality of user experiences in mobile systems. Moreover, unlike in HDDs, logical and physical fragmentation in flash storage must be handled in different manners.

Janusd is designed to effectively cope with the problems arising from logical and physical fragmentation at a low cost. Fig. 9 shows an organization of janusd with two defraggers, janusdL and janusdP, which are implemented as a user-level tool like e4defrag. Once the janusdL or janusdP is run by end users, it collects information of files to decide whether or not to trigger logical or physical defragmentation. To perform logical/physical fragmentation, special supports from the flash storage side are required. Those supportive functions are implemented as a firmware module, called janusdFTL, which is an extension of the existing FTL algorithm. Janusd is designed with a minimal change to the existing system. Thus, it is unnecessary to change the underlying file system and OS kernel, except for the

addition of a device driver for communication between the user-level tool and janusdFTL.

JanusdL is responsible for resolving logical fragmentation of files. JanusdL selects a list of fragmented files based on DoF^L of files (see ① in Fig. 9). Instead of physically moving files' data to another location, it sends a defragmentation command to janusdFTL (②) so that the logical-to-physical mapping table inside of flash storage (③) will be updated. This design enables us to resolve logical fragmentation without any physical data copies (see Section 4.1). JanusdP does not change logical layouts of files. Instead, it is in charge of resolving physical fragmentation for better exploitation of multiple channels in flash storage by re-distributing data among channels. JanusdP notifies janusdFTL of a list of frequently accessed files (④), and janusdFTL calculates the DoF^P values of the files (⑤) based on the physical data allocation inside of flash storage. Because data copies have negative impact on flash memory lifetime (see Section 4.2), among the frequently accessed files, janusdFTL performs physical defragmentation only on the files with high DoF^P values (⑥).

For the janusdL/P and janusdFTL to communicate with each other, new custom interfaces must be added. Table 2 summarizes a set of new custom interfaces, which can be implemented using user-defined command facilities of SATA and NVMe. Detailed descriptions of janusdL/P will be given in the following subsections.

4.1 JanusdL: Logical Defragmentation

Because janusdL inherits most of the features and algorithms from e4dfrag, the implementation of janusdL is done with slight modifications of e4dfrag.

Logical Defragmentation: When janusdL is invoked, it first searches for fragmented files using file-to-storage mapping. JanusdL calls the `FIBMAP` command of the Linux VFS to obtain a list of logical block addresses (LBAs) where the data of a given file is stored, and then it calculates the values of DoF^L of the file accordingly. With a list of files for logical defragmentation, the following process repeats for each of the files: JanusdL first looks for free and continuous LBAs as the destination where the file fragments can be moved to. These destination LBAs are obtained using the existing free-space allocation feature in e4dfrag. With the LBAs of the file fragments (source LBAs) and the destination LBAs, janusdL sends a `defrag` command, shown in Table 2, containing pairs of source-destination LBAs to janusdFTL in flash storage. Upon receiving `defrag` command, janusdFTL updates its logical-to-physical (L2P) mapping table so that the destination LBAs will refer to the physical pages referred to by the source LBAs. After completion of the command, janusdL revises the pointers in the inode of the defragmented file so that host applications can access the file through continuous LBAs.

Table 2: Custom interfaces for janusd.

Command	Description
defrag(list `src_LBA`, list `dst_LBA`)	Change `src_LBA` in logical-to-physical mapping table to `dst_LBA`.
flush()	Flush buffered defrag log to flash from DRAM.
check()	Check whether commit completion flag is saved at defrag log or not.
discard()	Delete the uncommitted log entries in defrag log.

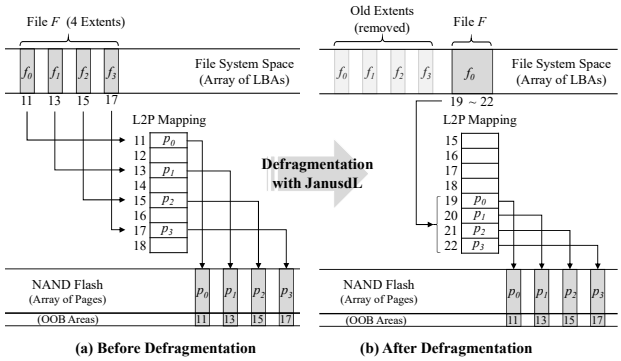


Fig. 10: An example of defragmentation in janusd.

Fig. 10 illustrates an example of how janusdL performs logical defragmentation. We assume that a target file F for defragmentation is fragmented into four extents f_0, f_1, f_2, and f_3, and they are stored in LBAs 11, 13, 15, and 17 (source LBAs), respectively. JanusdL sends a defrag command to map the extents to new LBAs 19 to 22 (destination LBAs). JanusdFTL first locates a list of physical pages that are mapped to the source LBAs. In this example, the file extents f_0, f_1, f_2, and f_3 at the source LBAs 11, 13, 15, and 17 are mapped to physical pages p_0, p_1, p_2, and p_3, respectively. JanusdFTL then updates the mapping entries of the destination LBAs 19 to 22 so that they refer to the physical pages p_0 to p_3, respectively. Finally, the L2P mapping entries of the source LBAs are unset, and janusdFTL sends an acknowledgment to the host to finish the defrag command. After this, `janusdL` revises the inode of the file to access the new extent f_0 through the new LBAs 19 to 22.

Power Failure Recovery: JanusdL may introduce inconsistency between L2P and P2L mapping information in the event of unexpected power failures. When new data is being written to a page, the FTL stores a corresponding LBA in an OOB area of that page for reverse P2L mapping. Even after a power failure occurs and an L2P mapping table (in DRAM) is lost, the FTL is able to recover a complete L2P mapping table by scanning all of the OOB areas in NAND flash. Unfortunately, when an L2P mapping table gets updated by janusd, corresponding LBAs in OOB areas cannot be updated in sync with the changes of L2P mapping because of NAND flash's erase-before-write constraint. In Fig. 10, for example, the LBA referring to the page p_0 was changed from 11 to 19, but the page p_0 still stores the old LBA (i.e., 11) in its OOB area. Suppose that the L2P mapping table is

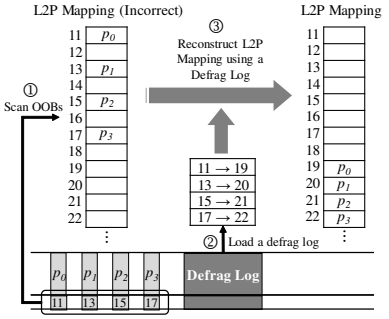


Fig. 11: A power failure recovery of janusdL.

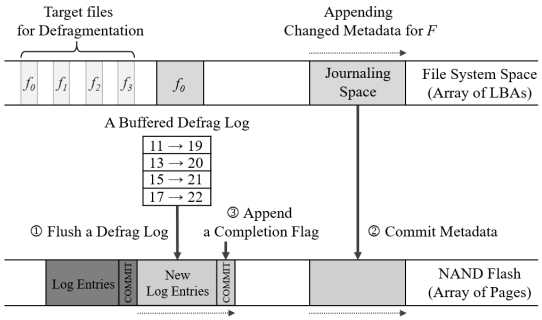


Fig. 12: A synchronization of file-system's metadata and defrag-log commits.

lost due to a power failure. The FTL will rebuild the L2P mapping table by scanning OOB areas. Based on the old P2L information in OOB areas, the page p_0 is referred to by LBA 11. However, at the file-system level, the new extent f_0 is at LBAs 19 to 22 because the inode of the file has been changed. As a result, when applications attempt to access f_0, the file-system sends wrong LBAs (e.g., 19) and the FTL returns invalid data or reports an error.

JanusdL addresses the inconsistency problem by logging all of the history of remapped LBAs in a special log, called a *defrag log*. A defrag log is an ordered collection of entries, each of which is a pair of a source LBA and a destination LBA plus a length. This information can easily be extracted from defrag commands. For example, a defrag log entry for f_0 is (11, 19, 1), where 11 is a source LBA, 19 is a destination LBA, and 1 is a length. Fig. 11 shows an example of how the mapping table is reconstructed after an unexpected power failure. When a flash storage device is rebooted, the FTL scans OOB areas of all pages and builds the L2P mapping table as usual. Then, it checks the defrag log to see if any L2P entries have been remapped for defragmentation and updates the mapping table accordingly.

To prevent frequent writes to flash, janusdFTL keeps defrag log entries in DRAM temporally and flushes them to flash at proper timings. This buffering, however, potentially causes another inconsistency problem – if a power failure occurs before the buffer is flushed to flash, the inconsistency between L2P and P2L mapping occurs. This problem can be solved by using a commit protocol combined with `fsck`. Fig. 12 illustrates how the commit protocol guarantees atomicity of defragmentation. Once all target files are moved and defragmentation is ready to finish, janusdL explicitly (1) flushes the buffered defrag log to flash by transmitting `flush` command in Table 2, (2) writes all file-system's metadata to a journaling area, and (3) appends a *commit completion flag* to the end of the defrag log. On system rebooting, `fsck` modified for janusdL first checks if the latest commit completion flag was written successfully by sending `check` command in Table 2. If not, the system was improperly shut down due to a system failure. Using `discard` command in Table 2, the modified `fsck` asks janusdFTL to discard uncommitted log entries in the defrag log and to rebuild an L2P mapping table only with committed ones. In the file system level, at the same time, the modified `fsck` rollbacks all the changes made to files by janusdL and reverts the files to their last consistent states.

Defrag Log Management: The FTL conducts internal page movements for garbage collection and wear leveling. If these page movements involve a page whose LBA is previously remapped, the defrag log must be updated. When a page is moved by garbage collection or wear leveling, janusdFTL writes the page according to its most recent P2L mapping information. The update of L2P mapping is required when a page is overwritten with new data as well. For both cases, since the P2L page mapping has been rewritten to flash, the corresponding old log entry should be removed.

Fig. 13 illustrates how janusdFTL manages the defrag log during garbage collection. Suppose that the flash block where valid pages p_0, p_1, p_2 and p_3 are stored is selected as a victim so that those pages are moved to four free pages p_4, p_5, p_6 and p_7, respectively. Accordingly, the L2P mapping table is updated to refer to new page locations. While moving valid pages, janusdFTL updates P2L mapping in OOBs if they are previously remapped by the defrag remapper. For example, 11 in p_0 is changed to 19 in p_4. After this, the entries of the moved pages are deleted from the defrag log. For example, entries (11, 19, 1), ..., (17, 22, 1) are now unnecessary. However, because of the overwrite restriction, janusdFTL has to append log entries to the defrag log, (11, Ø, 1), ..., (17, Ø, 1), to mark the old entries of LBAs 11 to 17 deleted. By this design, the defrag log may have multiple entries for the same LPAs, for example, (11, 19, 1) and (11, Ø, 1). To ignore old entries when the defrag log is scanned, janusdFTL writes a unique version number together.

As astute readers may notice, the defrag log would grow very large over time. To prevent this, janusdFTL sets a limit on the defrag log size. Once the size limit is reached, janusdFTL performs compaction – it selects flash blocks containing part of the defrag log, filters out obsolete entries, and writes only valid entries to the defrag log. (11, 19, 1) and (11, Ø, 1) are examples of obsolete entries – since L2P is equivalent to P2L, there is no need to keep them in the defrag log. The maximum

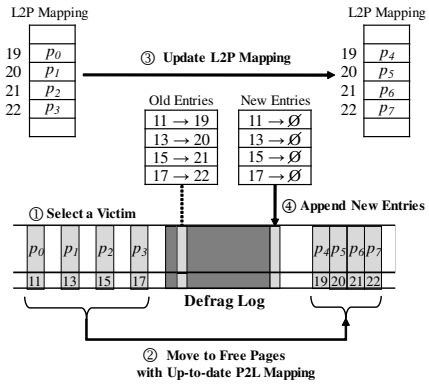


Fig. 13: Updating defrag log during garbage collection.

size of the defrag log is currently set to 10 MB, which is large enough to hold several millions of entries. Thanks to its huge size, almost all of the log entries become obsolete before being selected for compaction, and thus compaction involves few entry copy operations.

4.2 JanusdP: Physical Defragmentation

Different from janusdL, janusdP involves data copies for physical defragmentation. To minimize the negative impact of data copies on flash lifetime, janusdP performs physical defragmentation only on selected files that meet the following criteria: 1) they must be frequently accessed and 2) they must have high dragees of physical fragmentation (i.e., high DoF^P values).

To measure read frequencies of files, we implement a daemon program that keeps track of the total count of read accesses of files using the `inotify` feature provided by the Linux kernel. The read counts of files are stored in a separate file, and the janusdP utility reads the file to determine a list of 50 most frequently read files. JanusdP and janusdL use the same command to communicate with janusdFTL. To notify janusdFTL of physical defragmentation on a file, janusdP stores all the LBAs associated with the file as the source LBAs of a defrag command, but fills all the destination LBAs of the command with a null value -1. In this way, janusdFTL can easily distinguish a command for logical defragmentation from a command for physical defragmentation.

After janusdFTL receives a command for physical defragmentation, it first calculates the DoF^P value for the source LBAs stored in the defrag command. Recall that the DoF^P value associated with a set of LBAs is 0 if the LBAs can be accessed through the maximum I/O parallelism inside of flash storage. We employ 0.5 as an empirical threshold of DoF^P for janusdFTL to conduct physical defragmentation on the source LBAs. If the DoF^P of the LBAs is higher than or equal to 0.5, janusdFTL re-distributes the data (mapped to the source LBAs) among channels for the best I/O parallelism of future accesses. If the DoF^P of the LBAs is lower than 0.5, janusdFTL does nothing because the benefit of physical defragmentation would be marginal.

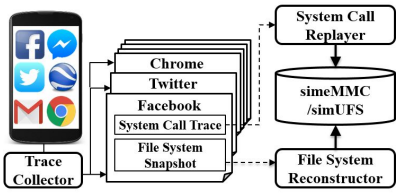


Fig. 14: An overview of our evaluation platform.

5 Experimental Results

In order to objectively understand the performance implication of janusd, we implement a comprehensive evaluation platform in the Linux operating system that supports three useful features, including (1) file-system snapshot/replication, (2) trace collection/replay, along with (3) mobile storage emulation. This evaluation platform makes it possible for us to conduct a set of the evaluations in an easy and convenient manner without modifying various smartphone platforms.

Fig. 14 illustrates our evaluation platform. The snapshot/replication tool allows us to take a storage snapshot of a smartphone and to replicate the same one in local flash storage for experiments. The trace collection/replay tool helps us to collect system-call events (e.g., `read()` and `write()`) from various applications running on real-world smartphones, and it replays them on the local storage. Those features enable us to repeat exactly the same I/O workloads on the same storage setup while varying defragmentation policies.

It is impossible to modify mobile storage devices like eMMC and UFS. Thus, we build two emulated mobile flash devices, called `simeMMC` and `simUFS`, using a customizable SSD device based on Samsung's 843T SSD [27]. 843T SSD supports extended SATA interfaces that allow a host system to directly control channels using NAND-specific I/O primitives (e.g., a page read/write and block erasure). Based on those interfaces, we implement a complete page-level FTL in a block layer of the Linux kernel (ver. 3.10). eMMC and UFS have similar channel architectures as conventional SSDs, except that they have smaller numbers of channels due to limited power budgets. We emulate I/O throughputs of eMMC and UFS by limiting the number of available channels of the 843T SSD to 4 and 8 for `simeMMC` and `simUFS`, respectively. To simulate a smaller I/O queue depth of mobile storage, we also intentionally increase end-to-end I/O latencies between the host and the flash device. As a result, both `simeMMC` and `simUFS` can accurately simulate I/O performance of eMMC and UFS devices over various request sizes.

As mentioned in Section 4, we implement janusdL/P as a user-level tool using e4defrag. The number of code lines newly added to e4defrag is about 400. janusdFTL is implemented as an extended module of the page-level FTL in the block layer. The custom interfaces between janusdL/janusdP and janusdFTL listed in Table 2 are implemented using the ioctl facility of the Linux.

5.1 Usage Scenario of Smartphone

We collect I/O activities of six popular applications running on N6. Table 3 summarizes the usage scenarios of each application. Each scenario starts with launching an application and runs specific tasks described in Table 3 for 10 minutes. The file system utilization is about 83%.

In order to perform evaluations under realistic environments, we create a six-month usage scenario of a smartphone. Based on a statistical study reporting that average daily time spent with a smartphone is 220 minutes [30], we simulate a daily usage scenario of a smartphone by repeating the six scenarios for 220 minutes. In a similar way, we finally create a six-month usage scenario by repeating the daily usage scenario 180 times. The applications are updated every 10 days based on the analysis of the update cycle of Android applications [28].

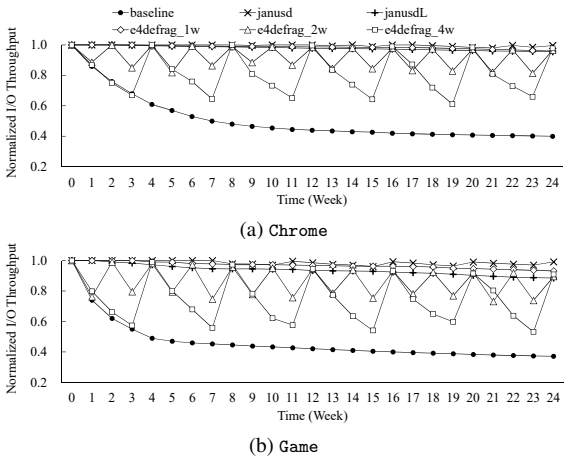
5.2 I/O Performance Analysis

While executing the six-month usage scenario, we compare the effect of six different defragmentation policies on performance: baseline, janusd, janusdL, e4defrag_1w, e4defrag_2w and e4defrag_4w. (Note that e4defrag_*n*w indicates when we invoke e4defrag with every *n* weeks.) For a fair comparison, before the execution of the scenario with a specific policy, the file system is initialized with the snapshot/replication tool mentioned in Section 5.1. Baseline does not perform file defragmentation. For janusd and janusdL, we execute janusd and janusdL every week. In the case of e4defrag, we invoke e4defrag with three different cycles, 1 week, 2 weeks and 4 weeks.

Fig. 15 shows that janusd achieves a consistent I/O throughput similar to or slightly better than e4defrag_1w ((a) Chrome 58 MB/s and (b) Game 66 MB/s). An interesting observation here is that the I/O throughput drops sharply even after one week without defragmentation. This indicates that frequent invocations of defragmentation are desirable to maintain high and consistent performance. In particular, janusd works better than janusdL and e4defrag_1w, offering the performance very close to the clean file system. Compared with janusdL and e4defrag_1w that perform only logical defragmentation, janusd conducts physical defragmentation that physically distributes fragmented pieces of files across different channels, improving I/O parallelism of file access. Fig. 16 shows I/O throughputs of the rest of the applications not shown in Fig. 15. On average, janusd improves the I/O throughput by 57% and 76% over baseline for simeMMC and simUFS, respectively. As expected, as the larger the values of DoF^L, the higher the I/O throughputs improved by janusd.

Table 3: A summary of benchmark scenarios.

Scenario	DoF^L	Scenario Description
Chrome	1.34	Launching app → Viewing webpages
Messenger	1.99	Launching app → Viewing chat records
Gmail	2.18	Launching app → Viewing emails
Facebook	2.55	Launching app → Viewing online news
Twitter	2.75	Launching app → Viewing online news
Game	3.02	Launching Lineage 2 → Playing game



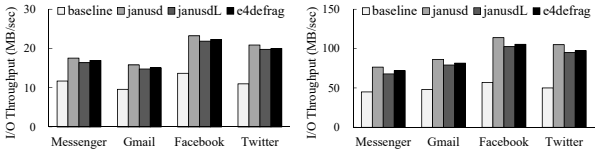
(a) Chrome

(b) Game

Fig. 15: Changes of I/O throughput over 6 months.

In order to analyze the impact of janusd on the quality of user experiences, we measure app launching times of the usage scenarios. We replay system call traces that are issued while an app is being launched, and then measure the reductions of I/O elapsed times spent by flash storage. Fig. 17 shows that janusd reduces the app launching times by up to 29% and 36% for simeMMC and simUFS over baseline, respectively. Our results confirm that janusd is effective in improving the quality of user experiences in smartphones.

Finally, Figs. 16 and 17 show that the performance improvement by janusd is more significant in a faster storage device like simUFS than a slower one, simeMMC. As observed in Section 3.2, the heavy fragmentation of files increases the number of small I/O requests to flash storage, which results in the increase of I/O stack overheads. SimUFS is more badly affected by the increased software I/O overheads – because of a fast storage access time, the handling of I/O requests at the software I/O stack level accounts for a larger proportion of the total I/O elapsed time. Janusd translates a large number of small I/Os to a fewer large ones, alleviating a performance penalty caused by I/O stack overheads. As a result, simUFS gets more benefits over simeMMC from the reduction of I/O stack overheads.



(a) simeMMC (b) simUFS

Fig. 16: The impact of janusd on the I/O throughput.

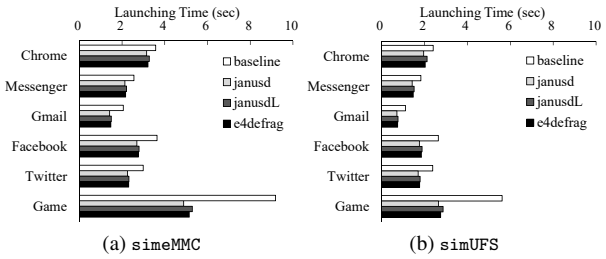


Fig. 17: The impact of janusd on the app launching time.

5.3 Lifetime Analysis

JanusdP has to physically move data. By performing physical defragmentation only on files that are physically fragmented and heavily read, janusdP minimizes its negative effect on flash lifetime. Table 4 shows that physical defragmentation by janusdP involves only a small amount of data copies, 364 MB, which is negligible compared to e4defrag_4w that copies data of 217 GB. Even though a smaller number of files are defragmented, its impact on performance is more significant than e4defrag as illustrated in Fig. 16. This is because janusdP optimally relocates files in multiple channels by taking into account the physical layout of flash storage.

Finally, we measure the amount of extra data movements needed for the maintenance of a defrag log in NAND flash. As mentioned in Section 4.2, we limit the size of a defrag log to 10 MB, and if its size exceeds the limit, janusd triggers compaction to reduce the log size. Since janusdL does not make data copies, the amount of data copies of janusdL in Table 4 indicates the amount data coped during the defrag log compaction. 219-MB data copies by janusdL is negligible over e4defrag_1w that involves 156-GB data copies for defragmentation.

6 Related Work

File Defragmentation: Recent interests in file defragmentation on flash storage were largely motivated by high-performance I/O support in flash storage. As flash storage gets faster, SW I/O stack overheads are emerging as a new I/O performance bottleneck, and flash fragmentation is reevaluated as a potential I/O bottleneck for flash storage. For example, Ji *et al.* showed that file fragmentation negatively affected the performance of mobile applications through an empirical study using several used smartphones [22]. In particular, they confirmed that redundant I/Os caused by fragmented files account for a nontrivial fraction of the total I/O time, degrading the overall I/O performance. More recently, Park *et al.* presented that file defragmentation on a log-structured file system reduced the frequency of I/O requests to a flash storage system, thereby improving the overall read performance [35]. While existing studies just discovered fragmentation problems [22-24] or presented a file-system-specific solution [35], our work, which is based on a detailed characterization study of flash file fragmentation, proposes a general scheme that can solve the fragmentation problem in flash storage, regardless of application types or system platforms.

Table 4: Impact of janusd on the amount of data copies.

e4defrag_1w	e4defrag_2w	e4defrag_4w	janusdL	janusdP
156 GB	182 GB	217 GB	0.219 GB	0.364 GB

Remapping Optimization in Flash: There are several studies proposed to improve flash storage performance by enhancing the remapping function of the FTL [31-34]. For example, Choi *et al.* presented a remapping technique that avoided double writing in journaling file systems [31]. Kang *et al.* proposed a transactional FTL for SQLite databases, which remapped a logical address from a physical location to a new physical location [32].

Our work is similar to the aforementioned studies in that it leverages an FTL's remapping function to offer better I/O performance. The above studies, however, did not take into account of the fragmentation problem in flash storage, and thus their remapping schemes could not effectively deal with fragmented files. Consequently, those studies are not applicable to resolve fragmentation.

7 Conclusions

We have presented a complete treatment for file fragmentation on mobile flash storage. From a systematic evaluation study, we showed that 1) file fragmentation is a recurring problem with a short recurrence interval and 2) the impact of file defragmentation on I/O performance is significant. By exploiting the decoupled fragmentation characteristics of flash storage, we proposed a novel flash-aware decoupled defragger, janusd, with two separate defraggers, janusdL and janusdP. JanusdL supports logical defragmentation without data copies by remapping the LBAs of the logically fragmented files with the FTL's mapping table. By saving a complete history of remapped LBA pairs in the defrag log, janusdL can safely recover from sudden power failures. On the other hand, janusdP, which is rarely invoked, improves the degree of the I/O parallelism of files which are severely limited in their available I/O parallelism. Our evaluation results showed that janusd can improve the I/O throughput by 57% and 76% on average in the Ext4 file systems on simeMMC and simUFS, respectively.

Our work can be extended in several directions. For example, janusdL can be easily extended to support different types of spatial locality of a file system such as free-space defragmentation. It would be also possible to support *defrag-on-write* that triggers logical defragmentation right after calling write() because the overhead of janusdL is negligible (i.e., $<$ 1 ms) over the cost of write() itself. Defrag-on-writes would realize a fragmentation-free file system, guaranteeing no performance degradation from fragmented files.

8 Acknowledgments

We would like to thank Ji-Yong Shin, our shepherd, and anonymous referees for valuable comments that greatly improved our paper. This research was supported by the National Research Foundation of Korea (NRF) grant funded by the Ministry of Science, ICT and Future Planning (MSIP) (NRF-2015M3C4A7065645), Ministry of Science and Technology of Taiwan (MOST 104-2221-E-009-011-MY3) and China National 863 Program 2015AA015304. The ICT at Seoul National University provided research facilities for this study. (*Corresponding Author: Jihong Kim*)

References

[1] MANTHUR, A., CAO, M., AND BHATTACHARYA, S. The New ext4 File System: Current Status and Future Plans. In *Proceedings of Linux Symposium* (2007).

[2] E4defrag - Online Defragmenter for Ext4 File System. `http://manpages.ubuntu.com/manpages/trusty/man8/e4defrag.8.html`

[3] Condusiv Diskeeper. `http://www.condusiv.com/products/diskeeper/`

[4] Auslogics Disk Defrag. `http://auslogics.com/en/software/disk-defrag/`

[5] Defraggler. `http://www.piriform.com/defraggler`

[6] Smart Defrag. `http://www.iobit.com/en/iobitsmartdefrag.php?a`

[7] Samsung SSD Performance Enhancement & Maintenance. `http://www.samsung.com/semiconductor/minisite/ssd/support/faqs-03.html`

[8] Frequently Asked Questions for Intel Solid State Drives. `http://www.intel.com/content/www/us/en/support/software/000006110.html`

[9] Crucial SSD and HDD Support & Maintenance. `http://www.crucial.com/usa/en/support-system-maintenance-defragment-hard-drive`

[10] KEHRER, O. O&O Defrag and Solid State Drives. `http://www.oo-software.com/en/docs/whitepaper/ood_ssd.pdf`

[11] LIND, A. Auslogics: How to Defrag Disk Drives The Right Way. `http://www.auslogics.com/en/articles/how-to-defrag/`

[12] Windows 8 TRIM SSD Instead of Defragmentation. `http://www.eightforums.com/tutorials/8615-optimize-drives-defrag-hdd-trim-ssd-windows-8-a.html`

[13] Windows 10 TRIM SSD Instead of Defragmentation. `http://www.tenforums.com/tutorials/8933-optimize-defrag-drives-windows-10-a.html`

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

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

[16] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M., AND PANIGRAHY, R. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference* (2008).

[17] KANG, J.-U., KIM, J.-S., PARK, C., PARK, H., AND LEE, J. A Multi-channel Architecture for High-performance NAND Flash-based Storage System. *Journal of Systems Architecture: the EUROMICRO Journal* (2007).

[18] PARK, S.-H., HA, S.-H., BANG, K., AND CHUNG, E.-Y. Design and Analysis of Flash Translation Layers for Multi-channel NAND Flash-based Storage devices. *IEEE Transactions on Consumer Electronics* (2009).

[19] HU, Y., JIANG, H., FANG, D., TIAN, L., AND LUO, H. Performance Impact and Interplay of SSD Parallelism Through Advanced Commands, Allocation Strategy and Data Granularity. In *Proceedings of the ACM International Conference on Supercomputing* (2011), pp. 96–107.

[20] JUNG, M., AND KANDEMIR, M. T. An Evaluation of Different Page Allocation Strategies on High-Speed SSDs. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems* (2012).

[21] JUNG, M., WILSON III, E. H., AND KANDEMIR, M. T. Physically Addressed Queueing (PAQ): Improving Parallelism in Solid State Disks. In *Proceedings of the International Symposium on Computer Architecture* (2012), pp. 404–415.

[22] JI, C., CHANG, L., SHI, L., WU, C., LI, Q., AND XUE, C. J. An Empirical Study of File-System Fragmentation in Mobile Storage Systems. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems* (2016).

[23] CONWAY, A., BAKSHI, A., JIAO, Y., ZHAN, Y., BENDER, M. A., JANNEN, W., JOHNSON, R., KUSZMAUL, B. C., PORTER, D. E., YUAN, J., AND FARACH-COLTON, M. File Systems Fated for Senescence? Nonsense, Says Science!. In *Proceedings of the USENIX Conference on File and Storage Technologies* (2017).

[24] KINSELLA, J. The Impact of Disk Fragmentation. http://condusiv.com/disk-defrag/fragmentation-impact/

[25] KESSLER, M. Maintaining Windows 2000 Peak Performance Through Defragmentation. https://msdn.microsoft.com/en-us/library/bb742585.aspx

[26] SINOFSKY, S. Disk Defragmentation Background and Engineering the Windows 7 Improvements. https://blogs.msdn.microsoft.com/e7/2009/01/25/disk-defragmentation-background-and-engineering-the-windows-7-improvements/

[27] SAMSUNG 843T Data Center Series. http://memorysolution.de/mso_upload/out/all/SM843T_Specification_v1.0.pdf

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

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

[30] HECHTEL, E. How Smartphones and Mobile Internet Have Changed Our Lives. https://testobject.com/blog/2016/01/smartphones-mobile-internet-changed-our-life.html

[31] CHOI, H.-J., LIM, S.-H., AND PARK, K.-H. JFTL: A Flash Translation Layer Based on A Journal Remapping for Flash Memory. *ACM Transactions on Storage* (2009).

[32] KANG, W.-H., LEE, S.-W., MOON, B., OH, G.-H., AND MIN, C. X-FTL: Transactional FTL for SQLite Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2013), pp. 97–108.

[33] ZHANG, Y., ARULRAJ, L. P., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. De-indirection for Flash-based SSDs with Nameless Writes. In *Proceedings of the USENIX Conference on File and Storage Technologies* (2012).

[34] OH, G., SEO, C., MAYURAM, R., KEE, Y., AND LEE, S. SHARE Interface in Flash Storage for Relational and NoSQL Databases. In *Proceedings of the International Conference on Management of Data* (2016), pp. 343–354.

[35] PARK, J., KANG, D.-H., AND EOM, Y.-I. File Defragmentation Scheme for A Log-structured File System. In *Proceedings of the ACM SIGOPS Asia-Pacific Workshop on Systems* (2016), pp. 19.

[36] TRAEGER, A. An Introduction to Linux Block I/O. http://researcher.ibm.com/researcher/files/il-AVISHAY/01-block_io-v1.3.pdf

[37] I/O Schedulers. http://www.makelinux.net/books/lkd2/ch13lev1sec5

[38] T10, TECHNICAL COMMITTEE OF THE INTERNATIONAL COMMITTEE ON INFORMATION TECHNOLOGY STANDARDS. SCSI TEST UNIT READY Command. http://www.t10.org/ftp/t10/document.06/06-022r0.pdf

[39] T10, TECHNICAL COMMITTEE OF THE INTERNATIONAL COMMITTEE ON INFORMATION TECHNOLOGY STANDARDS. SCSI Block Commands - 3 (SBC-3). http://www.t10.org/ftp/t10/document.05/05-344r0.pdf

[40] ANDERSON, D. C., CHASE, J. S., GADDE, S., GALLATIN, A. J., AND YOCUM, K. G. Cheating the I/O Bottleneck: Network Storage with Trapeze/Myrinet. In *Proceedings of the USENIX Annual Technical Conference* (1998).

[41] AHMAD, I., GULATI, A., AND MASHTIZADEH, A. vIC: Interrupt Coalescing for Virtual Machine Storage Device I/O. In *Proceedings of the USENIX Annual Technical Conference* (2011).

[42] HAHN, S.S. Impact of File Fragmentation on Android Smartphones. http://cares.snu.ac.kr/?view=publications&menuN=34#2017_Technical_Report

Octopus: an RDMA-enabled Distributed Persistent Memory File System

Youyou Lu
Tsinghua University

Jiwu Shu*
Tsinghua University

Youmin Chen
Tsinghua University

Tao Li
University of Florida

Abstract

Non-volatile memory (NVM) and remote direct memory access (RDMA) provide extremely high performance in storage and network hardware. However, existing distributed file systems strictly isolate file system and network layers, and the heavy layered software designs leave high-speed hardware under-exploited. In this paper, we propose an RDMA-enabled distributed persistent memory file system, Octopus, to redesign file system internal mechanisms by closely coupling NVM and RDMA features. For data operations, Octopus directly accesses a shared persistent memory pool to reduce memory copying overhead, and actively fetches and pushes data all in clients to re-balance the load between the server and network. For metadata operations, Octopus introduces self-identified RPC for immediate notification between file systems and networking, and an efficient distributed transaction mechanism for consistency. Evaluations show that Octopus achieves nearly the raw bandwidth for large I/Os and orders of magnitude better performance than existing distributed file systems.

1 Introduction

The in-memory storage and computing paradigm emerges as both HPC and big data communities are demanding extremely high performance in data storage and processing. Recent in-memory storage systems, including both database systems (e.g., SAP HANA [8]) and file systems (e.g., Alluxio [23]), have been used to achieve high data processing performance. With the emerging non-volatile memory (NVM) technologies, such as phase change memory (PCM) [34, 21, 46], resistive RAM (ReRAM), and 3D XPoint [7], data can be stored persistently in main memory level, i.e., *persistent memory*. New local file systems, including BPFS [11], SCMFS [42], PMFS [14], and HiNFS [32], are built recently to exploit the byte-addressability or persistence advantages of non-volatile memories. Their promising results have shown potentials of NVMs in high performance of both data storage and processing.

Meanwhile, the remote direct memory access (RDMA) technology brings extremely low latency and high bandwidth to the networking. We have measured an average latency and bandwidth of $0.9us$ and $6.35GB/s$ with a 56 Gbps InfiniBand switch, compared to $75us$ and $118MB/s$ with Gigabit Ethernet (GigaE). RDMA has greatly improved data center communications or RPCs in recent studies [13, 37, 19, 20].

Distributed file systems are trying to support RDMA networks for high performance, but mostly by substituting the communication module with an RDMA library. CephFS supports RDMA by using Accelio [2], an RDMA-based asynchronous RPC middleware. GlusterFS implements its own RDMA library for data communication [1]. NVFS [16] is a HDFS variant that is optimized with NVM and RDMA. And, Crail [9], a recent distributed file system from IBM, is built on the RDMA-optimized RPC library, DaRPC [37]. However, these file systems strictly isolate file system and network layers, by only replacing their data management and communication modules without refactoring the internal file system mechanisms. This layered and heavy software design prevents file systems from exploiting the hardware benefits. As we observed, GlusterFS has its software latency that accounts for nearly 100% on NVM and RDMA, while it is only 2% on disk. Similarly, it achieves only 15% of raw InfiniBand bandwidth, compared to 70% of the GigaE bandwidth. In conclusion, the *strict isolation* between the file system and network layers makes distributed file systems too heavy to exploit the benefits of emerging high-speed hardware.

In this paper, we revisit both data and metadata mechanism designs of the distributed file system by taking NVM and RDMA features into consideration. We propose an efficient distributed persistent memory file sys-

*Jiwu Shu is the corresponding author.

tem, Octopus[1], to effectively exploit the benefits of high-speed hardware. Octopus avoids the strict isolation of file system and network layers, and redesigns the file system internal mechanisms by closely coupling with NVM and RDMA features. For the data management, Octopus directly accesses a shared persistent memory pool by exporting NVM to a global space, avoiding stacking a distributed file system layer on local file systems, to eliminate redundant memory copies. It also rebalances the server and network loads, and revises the data I/O flows to offload loads from servers to clients in a client-active way for higher throughput. For the metadata management, Octopus introduces a self-identified RPC which carries sender's identifier with the RDMA write primitive for low-latency notification. In addition, it proposes a new distributed transaction mechanism by incorporating RDMA write and atomic primitives. As such, Octopus efficiently incorporates RDMA into file system designs that effectively exploit hardware benefits. Our major contributions are summarized as follows.

- We propose novel I/O flows based on RDMA for Octopus, which directly accesses a shared persistent memory pool without stacked file system layers, and actively fetches or pushes data in clients to rebalance server and network loads.
- We redesign metadata mechanisms leveraging RDMA primitives, including self-identified metadata RPC for low-latency notification, and a collect-dispatch distributed transaction for low-overhead consistency.
- We implement and evaluate Octopus. Experimental results show that Octopus effectively explores the raw hardware performance, and significantly outperforms existing RDMA-optimized distributed file systems.

2 Background and Motivation

2.1 Non-volatile Memory and RDMA

Non-Volatile Memory. Byte-addressable non-volatile memory (NVM) technologies, including PCM [34, 21, 46], ReRAM, Memristor [36], are being intensively studied in recent years. Intel and Micron have announced the 3D XPoint technology which is expected to be in product in the near future [7]. These NVMs have access latencies close to that of DRAM, while providing data persistence as hard disks. In addition, NVMs are expected to have better scalability than DRAM [34, 21]. Therefore, NVMs are promising candidates for storing data persistently at the main memory level.

[1]It is called Octopus because the file system performs remote direct memory access just like a Octopus uses its eight legs.

Remote Direct Memory Access. Remote Direct Memory Access (RDMA) enables low-latency network access by directly accessing memory from remote servers. It bypasses the operating system and supports zero-copy networking, and thus achieves high bandwidth and low latency in network accesses. There are two kinds of commands in RDMA for remote memory access:

(1) *Message Semantics*, with typical RDMA `send` and `recv` verbs for message passing, are similar to socket programming. Before sending an RDMA `send` request at the client side, an RDMA `recv` needs to be posted at the server side with an attached address indicating where to store the coming message.

(2) *Memory Semantics*, with typical RDMA `read` and `write` verbs, use a new data communication model (i.e., *one-sided*) in RDMA. In memory semantics, the memory address in remote server where the message will be stored is assigned at the sender side. This removes the CPU involvement of remote servers. The memory semantics provide relatively higher bandwidth and lower latency than the message semantics.

In addition, RDMA provides other verbs, including atomic verbs like `compare_and_swap` and `fetch_and_add` that enable atomic memory access of remote servers.

2.2 Software Challenges on Emerging High-Speed Hardware

In a storage system equipped with NVMs and RDMA enabled network, the hardware provides extremely higher performance than traditional media like hard disks and Gigabit Ethernet. Comparatively, overheads of the software layer, which are negligible compared to slow disk and Ethernet, now account for a significant part in the whole system.

Latency. To understand the latency overhead of existing distributed file systems, we perform synchronous 1KB write operations on GlusterFS, and collect latencies respectively in the storage, network, and software parts. The latencies are averaged with 100 synchronous writes. Figure 1(a) shows the latency breakdown of GlusterFS on disk (denoted as *diskGluster*) and memory (denoted as *memGluster*). To improve efficiency of GlusterFS on memory, we run memGluster on EXT4-DAX [4], which is optimized for NVM by bypassing the page cache and reducing memory copies. In diskGluster, the storage latency consumes the most part, nearly 98% of the total latency. In memGluster, the storage latency percentage drops dramatically to nearly zero. In comparison, the file system software latency becomes the dominate part, almost 100%. Similar trends have also been observed in previous studies in local storage systems [38]. While most distributed file systems stack the distributed data

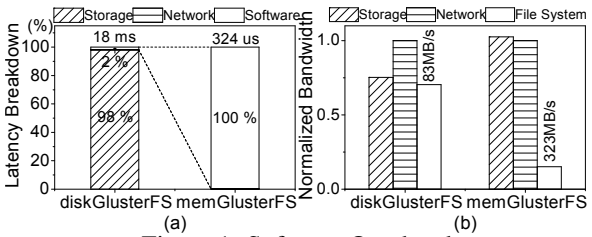


Figure 1: Software Overhead

management layer on another local file system (a.k.a, stacked file system layers), they face more serious software overhead than local storage systems.

Bandwidth. We also measure the maximum bandwidth of GlusterFS to understand the software overhead in terms of bandwidth. In the evaluation, we perform 1MB write requests to a single GlusterFS server repeatedly to get the average write bandwidth of GlusterFS. Figure 1(b) shows the GlusterFS write bandwidth against the storage and network bandwidths. In diskGluster, GlusterFS achieves a bandwidth that is 93.6% of raw disk bandwidth and 70.3% of raw Gigabit Ethernet bandwidth. In memGluster, GlusterFS's bandwidth is only 14.7% of raw memory bandwidth and 15.1% of raw InfiniBand bandwidth. Existing file systems are inefficient in exploiting the high bandwidth of new hardware.

We find that there are four mechanisms that contribute to this inefficiency in existing distributed file systems. First, data are copied multiple times in multiple places in memory, including user buffer, file system page cache, and network buffer. While this design is feasible for file systems that are built for slow disks and networks, it has a significant impact on system performance with high-speed hardware. Second, when networking is getting faster, the CPU at server side can be easily the bottleneck when processing requests from a lot of clients. Third, traditional RPC that is based on the event-driven model has relatively high notification latency when hardware provides low latency communication. Fourth, distributed file systems have huge consistency overhead in distributed transactions, owing to multiple network round-trips and complex processing logic.

As such, we propose to design an *efficient* distributed memory file system for high-speed network and memory hardware, by revisiting the internal mechanisms in both data and metadata management.

3 Octopus Design

To effectively explore the benefits of raw hardware performance, Octopus closely couples RDMA with file system mechanism designs. Both data and metadata mechanisms are reconsidered:

- **High-Throughput Data I/O**, to achieve high I/O bandwidth by reducing memory copies with a *Shared Persistent Memory Pool*, and improve throughput of small I/Os using *Client-Active I/Os*.
- **Low-Latency Metadata Access**, to provide a low-latency and scalable metadata RPC with *Self-Identified RPC*, and decrease consistency overhead using the *Collect-Dispatch Transaction*.

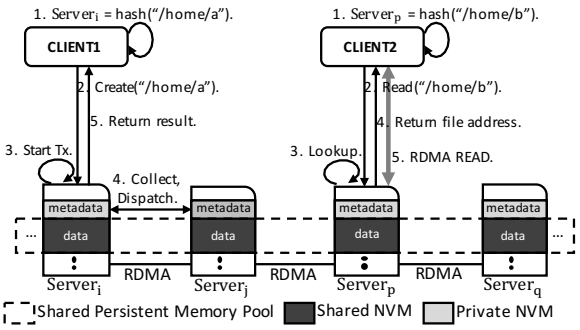


Figure 2: Octopus Architecture

3.1 Overview

Octopus is built for a cluster of servers that are equipped with non-volatile memory and RDMA-enabled networks. Octopus consists of two parts: *clients* and *data servers*. Octopus has no centralized metadata server, and the metadata service is distributed to different data servers. In Octopus, files are distributed to data servers in a hash-based way, as shown in Figure 2. A file has its metadata and data blocks in the same data server. But its parent directory and its siblings may be distributed to other servers. Note that the hash-based distribution of file or data blocks is not a design focus of this paper. Hash-based distribution may lead to difficulties in wear leveling issue in non-volatile memory, and we leave this problem for future work. Instead, we aim to discuss novel metadata and data mechanism designs that are enabled by RDMA in this paper.

In each server, the data area is exported and shared in the whole cluster for remote direct data accesses, while the metadata area is kept private for consistency reasons. Figure 3 shows the data layout of each server, which is organized into six zones: (1) *Super Block* to keep the metadata of the file system. (2) *Message Pool* for the metadata RPC for temporary message storage when exchanging messages. (3) *Metadata Index Zone* using a chained hash table to index the file or directory metadata nodes in the metadata zone. Each entry in the chained hash table contains `name`, `i_addr`, and `list_ptr` fields, which respectively represent the name of the file, the physical address of the file's inode, and the pointer to link the metadata index for the files that has a same hash value. A file hashes its name and locates its metadata index to fetch its `inode` address. (4) *Metadata Zone* to keep the file or directory metadata nodes (i.e., `inode`), each of which consumes 256 bytes. With the

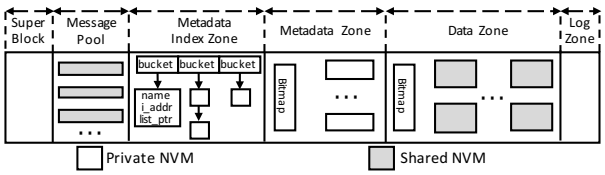


Figure 3: Data Layout in a Octopus Node

`inode`, Octopus locates the data blocks in the data zone. (5) *Data Zone* to keep data blocks, including directory entry blocks and file data blocks. (6) *Log Zone* for transaction log blocks to ensure file system consistency.

While a data server keeps metadata and data respectively in the private and shared area, Octopus accesses the two areas remotely in different ways. For the private metadata accesses, Octopus uses optimized remote procedure calls (RPC) as in existing distributed file systems. For the shared data accesses, Octopus directly reads or writes data objects remotely using RDMA primitives.

With the use of RDMA, Octopus removes duplicated memory copies between file system images and memory buffers by introducing the *Shared Persistent Memory Pool* (*shared pool* for brevity). This shared pool is formed with exported data areas from each data server in the whole cluster (in Section 3.2.1). In current implementation, the memory pool is initialized using a static XML configuration file, which stores the pool size and the cluster information. Octopus also redesigns the read/write flows by sacrificing network round-trips to amortize server loads using *Client-Active I/Os* (in Section 3.2.2).

For metadata mechanisms, Octopus leverages RDMA write primitives to design a low-latency and scalable RPC for metadata operations (in Section 3.3.1). It also redesigns the distributed transaction to reduce the consistency overhead, by collecting data from remote servers for local logging and then dispatching them to remote sides (in Section 3.3.2).

3.2 High-Throughput Data I/O

Octopus introduces a shared persistent memory pool to reduce data copies for higher bandwidth, and actively performs I/Os in clients to rebalance server and network overheads for higher throughput.

3.2.1 Shared Persistent Memory Pool

In a system with extremely fast NVM and RDMA, memory copies account for a large portion of overhead in an I/O request. In existing distributed file systems, a distributed file system is commonly layered on top of local file systems. For a read or write request, a data object is duplicated to multiple locations in memory, such as kernel buffer (`mbuf` in TCP/IP stack), user buffer (for storing distributed data objects as local files), kernel

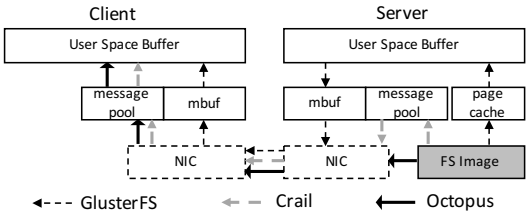


Figure 4: Data Copies in a Remote I/O Request

page cache (for local file system cache), and file system image in persistent memory (for file storage in a local file system in NVM). As the GlusterFS example shown in Figure 4, a remote I/O request requires the fetched data to be copied seven times including in memory and NIC (network interface controller) for final access.

Recent local persistent file systems (like PMFS [14] and EXT4-DAX [4]) directly access persistent memory storage without going through kernel page cache, but it does not solve problems in the distributed file systems cases. With direct access of these persistent memory file systems, only page cache is bypassed, and a distributed file system still requires data to be copied six times.

Octopus introduces the *shared persistent memory pool* by exporting the data area of the file system image in each server for sharing. The shared pool design not only removes the stacked file system design, but also enables direct remote access to file system images without any caching. Octopus directly manages data distribution and layout of each server, and does not rely on a local file system. Direct data management without stacking file systems is also taken in Crail [9], a recent RDMA-aware distributed file system built from scratch. Compared to stacked file system designs like GlusterFS, data copies in Octopus and Crail do not need to go through user space buffer in the server side, as shown in Figure 4.

Octopus also provides a global view of data layout with the shared pool enabled by RDMA. In a data server in Octopus, the data area in the non-volatile memory is registered with *ibv_reg_mr* when the data server joins, which allows the remote direct access to file system images. Hence, Octopus removes the use of a `message pool` or a `mbuf` in the server side, which are used for preparing file system data for network transfers. As such, Octopus requires data to be copied only four times for a remote I/O request, as shown in Figure 4. By reducing memory copies in non-volatile memories, data I/O performance is significantly improved, especially for large I/Os that incur fewer metadata operations.

3.2.2 Client-Active Data I/O

For data I/O, it is common to complete a request within one network round-trip. Figure 5(a) shows a read example. The client issues a read request to the server, and the server prepares data and sends it back to the client.

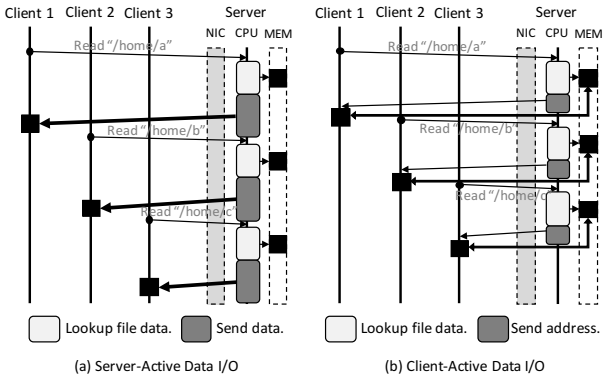


Figure 5: Comparison of Server-Active and Client-Active Modes

Similarly, a write request can also complete with one round-trip. This is called *Server-Active Mode*. While this mode works well for slow Ethernet, we find that the server is always in high utilization and becomes a bottleneck when new hardware is equipped.

In remote I/Os, the throughput is bounded by the lower one between the network and server throughput. In our cluster, we achieve 5 *million* network IOPS for 1KB writes, but have to spend around 2*us* (i.e., 0.5 *million*) for data locating even without data processing. The server processing capacity becomes the bottleneck for small I/Os when RDMA is equipped.

In Octopus, we propose *client-active mode* to improve server throughput by sacrificing the network performance when performing small size I/Os. As shown in Figure 5(b), in the first step, a client in Octopus sends a *read* or *write* request to the server. In the second step, the server sends back the metadata information to the client. Both the two steps are executed for metadata exchange using the self-identified metadata RPC which will be discussed next. In the third step, the client reads or writes file data with the returned metadata information, and directly accesses data using RDMA `read` and `write` commands. Since RDMA `read` and `write` are one-sided operations, which access remote data without participation of CPUs in remote servers, the server in Octopus has higher processing capacity. By doing so, a rebalance is made between the server and network overheads. With introduced limited round-trips, server load is offloaded to clients, resulting in higher throughput for concurrent requests.

Besides, Octopus uses the per-file read-write lock to serialize the concurrent RDMA-based data accesses. The lock service is based on a combination of GCC (GNU Compiler Collection) and RDMA atomic primitives. To read or write file data, the locking operation is executed by the server locally using GCC atomic instructions. The unlock operation is executed remotely by the client with RDMA atomic verbs after data I/Os. Note that serializability between GCC and RDMA atomic primitives is not guaranteed due to lack of atomicity between the CPU and the NIC [10, 41, 19]. In Octopus, GCC and RDMA atomic instructions are respectively used in the locking and unlocking phases. This isolation prevents the competition between the CPU and the NIC, and thus ensures correctness of parallel accesses.

3.3 Low-Latency Metadata Access

RDMA provides microsecond level access latencies for remote data access. To explore this benefit in the file system level, Octopus refactors the metadata RPC and distributed transaction by incorporating RDMA write and atomic primitives.

3.3.1 Self-Identified Metadata RPC

RPCs are used in Octopus for metadata operations. Both message and memory semantic commands can be utilized to implement RPCs.

(1) *Message-based RPC.* In the message-based RPC, a `recv` request is firstly assigned with a memory address, and then initialized in the remote side before the `send` request. Each time an RDMA `send` arrives, an RDMA `recv` is consumed. Message-base RPC has relatively high latency and low throughput. `send/recv` in UD (Unreliable Datagram) mode provides higher throughput [20], but is not suitable for distributed file systems due to its unreliable connections.

(2) *Memory-based RPC.* RDMA `read/write` have lower latency than `send/recv`. Unfortunately, these commands are one-sided, and remote server is uninvolved. To timely process these requests, the server side needs to scan the message buffers repeatedly to discover new requests. This causes high CPU overhead. Even worse, when the number of clients increased, the server side needs to scan more message buffers, and this in turn increases the processing latency.

To gain benefits of both sides, we propose the self-identified metadata RPC. Self-identified metadata RPC attaches the sender's identifier with the RDMA write request using the RDMA `write_with_imm` command. `write_with_imm` is different from RDMA `write` in two aspects: (1) it is able to carry an immediate field in the message, and (2) it notifies remote side immediately, but RDMA `write` does not. With the first difference, we attach the client's identifier in the immediate data field including both a `node_id` and an `offset` of the client's receive buffer. For the second difference, RDMA `write_with_imm` consumes one receive request from the remote queue pair (QP), and thus gets immediately processing after the request arrives. The identifier attached in the immediate field helps the server to direct locate the new message without scanning the whole buffer. After

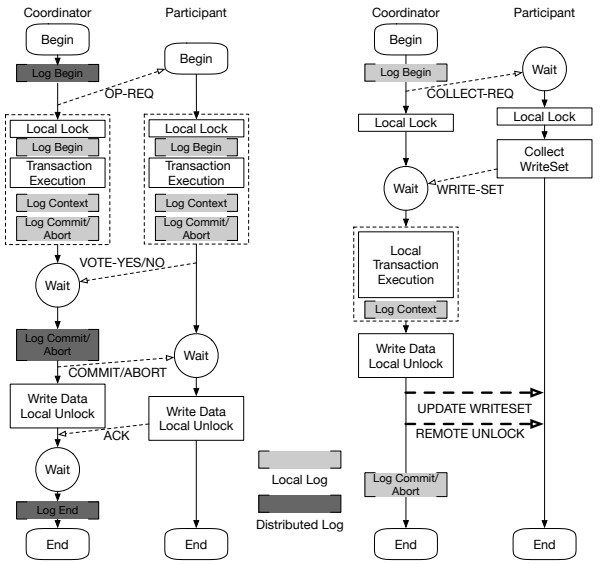


Figure 6: Distributed Transaction

processing, the server uses RDMA `write` to return data back to the specified address of `offset` in the client of `node_id`. Compared to buffer scanning, this immediate notification dramatically lowers down the CPU overhead when there are a lot of client requests. As such, the self-identified metadata RPC provides low-latency and scalable RPCs than `send/recv` and `read/write` approaches.

3.3.2 Collect-Dispatch Transaction

A single file system operation, like *mkdir*, *mknod*, *rmnod* and *rmdir* in Octopus, performs updates to multiple servers. Distributed transactions are needed to provide concurrency control for simultaneous requests and crash consistency for the atomicity of updates across servers. The two-phase commit (2PC) protocol is usually used to ensure consistency. However, 2PC incurs high overhead due to its distributed logging and coordination for both locks and log persistence. As shown in Figure 6(a), both locking and logging are required in coordinator and participants, and complex network round-trips are needed for negotiation for log persistence ordering.

Octopus designs a new distributed transaction protocol named *Collect-Dispatch Transaction* leveraging RDMA primitives. The key idea lies in two aspects, respectively in crash consistency and concurrency control. One is **local logging with remote in-place update** for crash consistency. As shown in Figure 6(b), in collect phase, Octopus collects the read and write sets from participants, and performs local transaction execution and local logging in the coordinator. Since participants do not need to keep logging, there is no need for complex negotiation for log persistence between coordinator and participants, thereby reducing protocol overheads. For the dispatch phase, the coordinator spreads the updated write set to the participants using RDMA `write` and releases the corresponding lock with RDMA atomic primitives, without the involvements of the participants.

The other is **a combination of GCC and RDMA locking** for concurrency control, which is the same as the lock design in the data I/Os in Section 3.2.2. In collect-dispatch transactions, locks are added locally using the GCC `compare_and_swap` command in both coordinator and participants. For the unlock operations, the coordinator releases the local lock using the GCC `compare_and_swap` command but the remote lock in each participant using the RDMA `compare_and_swap` command. The RDMA unlock operations do not involve the CPU processing of participants, and thus simplify the unlock phase.

As a whole, collect-dispatch requires one RPC, one RDMA write, and one RDMA atomic operation, and 2PC requires two RPCs. Collect-Dispatch still has lower overhead, because (1) RPC has higher latency than an RDMA write/atomic primitive, (2) RDMA write/atomic primitive does not involve CPU processing of remote side. Thus, we conclude collect-dispatch is efficient, as it not only removes complex negotiations for log persistence ordering across servers, but reduces costly RPC and CPU processing overheads.

Consistency Discussions. In persistent memory systems, data cache in the CPU cache needs to be flushed to the memory timely and ordered to provide crash consistency [11, 26, 33, 25, 14, 32]. In Octopus, metadata consistency is guaranteed by the collect-dispatch transaction, which uses `clflush` to flush data from the CPU cache to the memory to force persistence of the log. While the collect-dispatch transaction can be used to provide data consistency, data I/Os are not wrapped in a transaction in current Octopus implementation for efficiency. We expect that RDMA will have more efficient remote flush operations that could benefit data consistency, such as novel I/O flows like RDMA read for remote durability [12], new proposed commands like `RDMA commit` [39], or new designs that leverage availability for crash consistency [45]. We leave efficient data consistency for future work.

4 Evaluation

In this section, we evaluate Octopus's overall data and metadata performance, then the benefits from each mechanism design, and finally its performance for big data applications.

4.1 Experimental Setup

Evaluation Platform. In the evaluation, we run Octopus on servers with large memory. Each server is equipped

with 384GB DRAM and two 2.5GHz Intel Xeon E5-2680 v3 processors, and each processor has 24 cores. Clients run on different servers. Each client server has 16GB DRAM and one Intel Xeon E2620 processor. All these servers are connected with a Mellanox SX1012 switch using CX353A ConnectX-3 FDR HCAs (which support 56 Gbps over InfiniBand and 40GigE). All of them are installed with Fedora 23.

Evaluated File Systems. Table 1 lists the distributed file system (DFSs) for comparison. All these file systems are deployed in memory of the same cluster. For existing DFSs that require local file systems, we build local file systems on DRAM with *pmem* driver and *DAX* [5] supported in *ext4*. The EXT4-DAX [4] is optimized for NVM which bypasses the page cache and reduces memory copies. Octopus manages its storage space on the emulated persistent memory using shared memory (SHM) of Linux in each server. These file systems are allocated with 20GB for file system storage at each server. For the network part, all distributed file systems run on RDMA directly. Specifically, memGluster supports using RDMA protocol for communication between glusterfs clients and glusterfs bricks. NVFS is an optimized version of HDFS which exploits the advantages of byte-addressability of NVM and RDMA. Crail is a recent open-source DFS from IBM, and it relies on DaRPC [37] for RDMA optimization and reserves huge pages as transfer cache for bandwidth improvement.

Table 1: Evaluated File Systems

memGluster	GlusterFS runs on memory, and GlusterFS is a widely-used DFS that has no centralized metadata services and is now a part of Redhat
NVFS [16]	a version of HDFS that is optimized with both RDMA and NVM
Crail [9]	an in-memory RDMA-optimized DFS built with DaRPC [37]
memHDFS [35]	HDFS runs on memory, and HDFS is a widely-used DFS for big data processing
Alluxio[23]	an in-memory file system for big data processing

Workloads. In our evaluation, we compare Octopus with memGluster, NVFS and Crail for metadata and read-write performance, and compare it with NVFS and Alluxio for big data benchmarks. We use *mdtest* for metadata evaluation, *fio* for read/write evaluation, and an in-house read/write tool based on *openMPI* for aggregated I/O performance. For big data evaluation, we replace HDFS by adding Octopus plugin under Hadoop. We use three package-in MapReduce benchmarks in Hadoop, i.e., TestDFSIO, Teragen, and Wordcount, for evaluation.

4.2 Overall Performance

To evaluate Octopus, we first compare its overall performance with memGluster, NVFS and Crail. All these file systems are running in the memory level with RDMA-enabled InfiniBand network. In this evaluation, we first compare Octopus's latency and bandwidth to the raw network's and storage's latency and bandwidth, and then compare Octopus's metadata and data performance to other file systems.

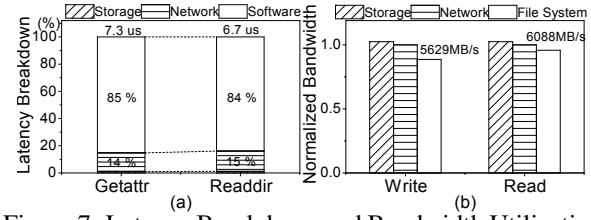


Figure 7: Latency Breakdown and Bandwidth Utilization

4.2.1 Latency and Bandwidth Breakdown

Figure 7 shows both single round-trip latency and bandwidth breakdown for Octopus. From the figures, we have two observations.

(1) The software latency is dramatically reduced to 6*us* (around 85% of the total latency) in Octopus, from 323*us* (over 99%) in memGluster, as shown in Figure 7(a). For the memGluster on the emerging non-volatile memory and RDMA hardwares, the file system layer has a latency that is several orders larger than that of storage or network. The software consumes the overwhelmed part, and becomes a new bottleneck of the whole storage system. In contrast, Octopus is effective in reducing the software latency by redesigning the data and metadata mechanisms with RDMA. The software latency in Octopus is in the same order with the hardware.

(2) Octopus achieves read/write bandwidth that approaches the raw network bandwidth, as shown in Figure 7(b). The raw storage and network bandwidths respectively are $6509MB/s$ (with single-thread memcpy) and $6350MB/s$. Octopus achieves a read/write ($6088/5629MB/s$) bandwidth that is 95.9%/88.6% of the network bandwidth. In conclusion, Octopus effectively exploits the hardware bandwidth.

4.2.2 Metadata Performance

Figure 8 shows the file systems' performance in terms of metadata IOPS with different metadata operations by varying the number of data servers. From the figure, we make two observations.

(1) Octopus has the highest metadata IOPS among all evaluated file systems in general. memGluster and NVFS provide metadata IOPS in the order of 10^4. Crail provides metadata IOPS in the order of 10^5 owing to DaRPC, a high performance RDMA-based RPC. Comparatively, Octopus provides metadata IOPS in the order of 10^6, which is two orders higher than memGluster and NVFS. Octopus achieves the highest throughput

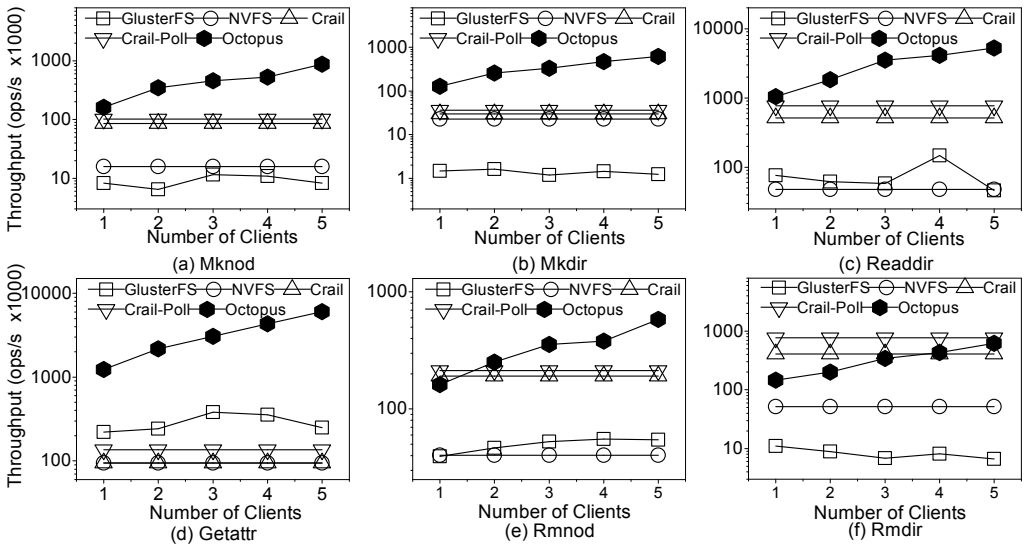


Figure 8: Metadata Throughput

except for *rmdir* and *rmnod* when there is only one data server. Crail is slightly better in this case, because it is deployed with *RdmaDataNode* mode without transaction guarantee. Generally, Octopus achieves high throughput in processing metadata requests, which mainly owes to the self-identified RPC and collect-dispatch transaction that promise extremely low latency and high throughput.

(2) Octopus achieves much better scalability than the other evaluated file systems. NVFS and Crail are designed with single metadata server, and achieve constant metadata throughput. Even with one metadata server, Octopus achieves better throughput than these two file systems in most cases. memGluster achieves the worst throughput, for GlusterFS is designed to run on hard disks and the software layer is inefficient in exploring the high performance of NVM and RDMA, which has been illustrated in Section 2.2. Besides, memGluster stacks its data management layer on top of the local file system in each server to process metadata requests, and this also limits the throughput. Comparatively, Octopus has the best scalability. For all evaluated metadata operations, Octopus's IOPS is improved by 3.6 to 5.4 times when the number of servers is increased from 1 to 5.

4.2.3 Read/Write Performance

Figure 9 shows the file systems' performance in terms of concurrent read/write throughput with multiple clients by varying the read/write sizes. From figure 9, we can see that, with small read/write sizes, Octopus achieves much higher throughput than other file systems (750 Kops/s and 1 Mops/s for writes and reads respectively). This benefit mainly comes from the client-active data I/O and self-identified RPC mechanisms. NVFS achieves relatively high throughput when read/write size is set to 1KB, for its buffer manager prefetches data to boost

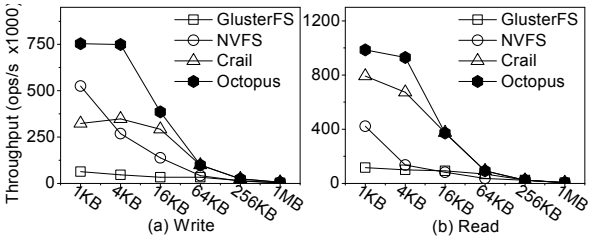


Figure 9: Data I/O Throughput (Multiple Clients)

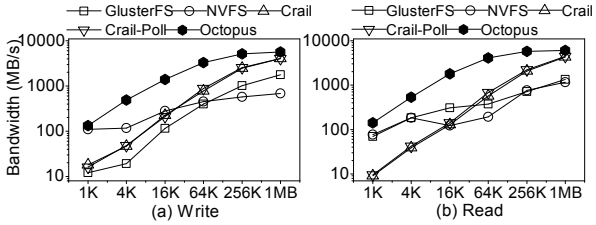


Figure 10: Data I/O Bandwidth (Single Client)

performance. But it drops rapidly when the I/O size grows, which is mainly restricted by the performance of RPC efficiency. Crail has lower throughput than NVFS when I/O size is small, but it achieves throughput close to Octopus when I/O size grows. memGluster has the worst throughput and only achieves 100 Kops/s.

Figure 10 shows the read/write bandwidth achieved by a single client with different read/write sizes. As shown in the figure, Octopus significantly outperforms existing DFSs in terms of read or write bandwidth. When the I/O size is set to 1MB, the read/write bandwidths in NVFS and memGluster are around only $1000MB/s$ and $1500MB/s$, respectively. Crail reaches a bandwidth of $4000MB/s$, which only occupies 63% of the raw network bandwidth. In contrast, Octopus can achieve bandwidth close to that of the raw InfiniBand network ($6088MB/s$ and $5629MB/s$ with 1MB I/O size for read and write respectively), which is mainly because of reduced memory copies by using a shared persistent memory pool.

4.3 Evaluation of Data Mechanisms

4.3.1 Effects of Reducing Data Copies

Octopus improves data transfer bandwidth by reducing memory copies. To verify the effect of reducing data copies, we implement a version of Octopus which add an extra copy at client side, and we refer to it as Octopus+copy. As shown in Figure 11, when I/O size is set to 1MB, Octopus+copy achieves nearly the same bandwidth as Crail (around $4000MB/s$). However, when the extra data copy is removed, Octopus can provide $6000MB/s$ of bandwidth that is written or read by a single client, 23% of extra bandwidth gained. When the I/O size is small, Octopus+copy still surpasses Crail with higher bandwidth, owing to closely coupled RDMA and file system mechanism designs to be evaluated next.

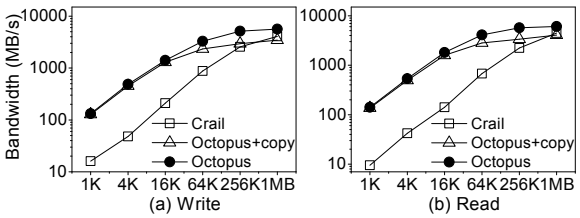


Figure 11: Effects of Reducing Data Copies

4.3.2 Effects of Client-Active Data I/O

We then compare the IOPS of data I/O in client-active and server-active modes that are mentioned in Section 3. Figure 12 shows the read/write throughput of both client-active and server-active modes of Octopus by varying read/write sizes. Crail's performance is also given for reference. We observe that the client-active mode has higher data throughput than the server-active mode for small read/write sizes. Both modes have close throughput for read/write sizes that are larger than 16KB. When the read/write sizes are smaller than 16KB, the client-active mode has higher data throughput by 193% for writes and 27.2% for reads on average. Even the client-active mode consists more network round-trips, it is more efficient to offload workloads to clients from servers when the read/write size is small, in order to improve the data throughput. Client-active mode improves write throughput more obviously than read throughput, because the server side has higher overhead for writes than reads in server-active mode. In server-active mode, after the server side reads data from the client using RDMA `read` when processing client's write operation, it has to check the completion of this operation, which is time-consuming. But for client's read operations, server side never checks the completion message, and provides relatively higher throughput. In all, we conclude that client-active mode has higher bandwidth than the commonly-used server-active mode.

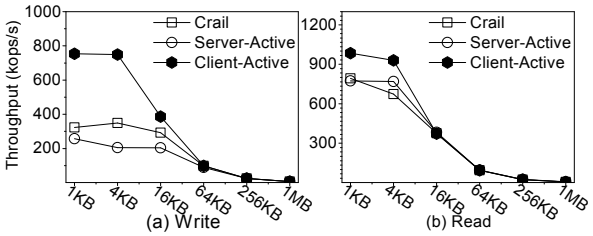


Figure 12: Client-Active Data I/O Performance

4.4 Evaluation of Metadata Mechanisms

4.4.1 Effects of Self-Identified Metadata RPC

We first compare raw RPC performance with different usage of RDMA primitives to evaluate the effects of self-identified metadata RPC. We then compare Octopus with existing file systems on metadata latencies.

Figure 13(a) shows the raw RPC throughput using three RPC implementations (i.e., message-based, memory-based, and self-identified, without message batch) along with DaRPC by varying the I/O sizes. DaRPC used in Crail is designed based on RDMA `send/recv`, and it achieves the lowest throughput, $2.4Mops/s$ with an I/O size of 16 bytes. Its performance may be limited by the Java implementation in its jVerbs interface. We also implement a message-based RPC that uses RDMA `send/recv` verbs, and it achieves a throughput of $3.87Mops/s$ at most. This throughput is limited by the raw performance of RDMA `send/recv`. For the memory-based RPCs that use RDMA `write` verbs, as taken in FaRM [13], we compare the performance by setting the maximum number of client threads to 20 and 100. As observed, the throughput is the highest (i.e., $5.4Mops/s$) when the maximum number of client threads is 20. However, it decreases quickly to $3.46Mops/s$ when the maximum number of client threads is 100. This shows the inefficiency in processing and notification in the memory-based RPCs when there are a large number of client threads. Our proposed self-identified RPC, which carry on client identifiers with the RDMA `write_with_imm` verbs, keeps constant high throughput for an average of $5.4Mops/s$, without being affected by the number of client threads. Similarly, we also measure the latency of each RPC (in Figure 13(b)), among which self-identified RPC keeps relative low latency. As such, self-identified RPCs provide scalable and low-latency accesses, which is suitable for distributed storage systems to support a large number of client requests.

Figure 14 shows metadata latencies of Octopus along with other file systems. As shown in the figure, Octopus achieves the lowest metadata latencies among all the evaluated file systems for all evaluated metadata operations (i.e., $7.3us$ and $6.7us$ respectively for `getattr`

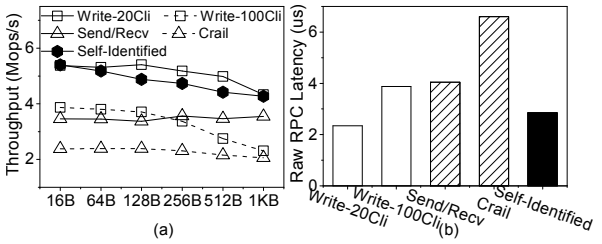


Figure 13: Raw RPC Performance

and `readdir`), which are close to the InfiniBand network latency for most cases. With the self-identified metadata RPC, Octopus can support low-latency metadata operations even without client cache. Crail uses DaRPC for inter-server communication. However, Crail's metadata (e.g., `mkdir` and `mknod`) latencies are much higher than raw DaRPC's latency. This possibly is because Crail is implemented on the inefficient HDFS framework, or it registers memory temporarily for message communication, which is time-consuming. NVFS and memGluster suffer the similar problem of heavy file system designs as Crail, and thus have relatively higher latency.

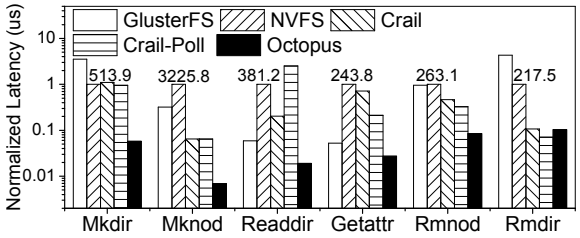


Figure 14: Metadata Latency

4.4.2 Effects of Collect-Dispatch Transaction

To evaluate the effects of the collect-dispatch transaction in Octopus, we also implement a transaction system based on 2PC for comparison. Figure 15(a) exhibits the latencies of these two transaction mechanisms. Collect-dispatch reduces latency by up to 37%. This is because 2PC involves two RPCs to exchange messages from remote servers, while collect-dispatch only needs one RPC and two one-sided RDMA commands to finish the transaction. Although the number of messages is increased, the total latency drops. RPC protocol needs the involvements of both local and remote nodes, and a lot of side information (e.g., hash computing, and message discovery) needs to be processed at this time. Thus, RPC latency (around 5*us*) is much higher than one-sided RDMA primitives (less than 1*us*). From figure 15(b) we can see that, transaction based on collect-dispatch improves throughput by up to 79%. On one hand, collect-dispatch only writes logs locally, significantly reducing logging overhead. On the other hand, collect-dispatch decreases the total number of RPC when processing transactions, which reduces the involvements of remote CPUs and thereby improves performance.

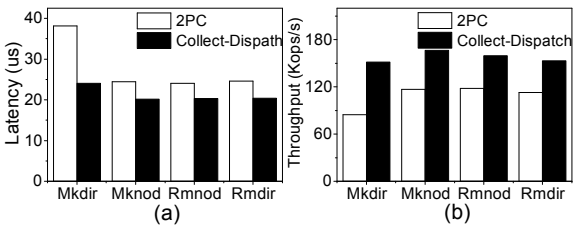


Figure 15: Collect-Dispatch Transaction Performance

4.5 Evaluation using Big Data Applications

In addition, we compare Octopus with distributed file systems that are used in big data framework. We configure Hadoop with different distributed file systems - memHDFS, Alluxio, NVFS, Crail and Octopus. In this section, we compare both read/write bandwidth and application performance.

Read/Write Bandwidth. Figure 16(a) compares the read/write bandwidths of above-mentioned file systems using *TestDFSIO* by setting the read/write size to 256KB. Octopus and Crail show much higher bandwidth than traditional file systems. Octopus achieves $2689MB/s$ and $2499MB/s$ for write and read operations respectively, and Crail achieves $2424MB/s$ and $2215MB/s$ respectively. Note that they have lower bandwidths than the results in `fio`. The reason is that we connect Octopus/Crail with Hadoop plugin using JNI (Java Native Interface), which restricts the bandwidth. In contrast, memHDFS, Alluxio and NVFS show lower bandwidth than Octopus and Crail. memHDFS has the lowest bandwidth, for the heavy HDFS software design that is for hard disks and traditional Ethernet. Alluxio and NVFS are optimized to run on DRAM, and thus provide higher bandwidth than memHDFS. But they are still slower than Octopus. Thus, we conclude the general-purpose Octopus can also be integrated into existing big data framework and provide better performance than existing file systems.

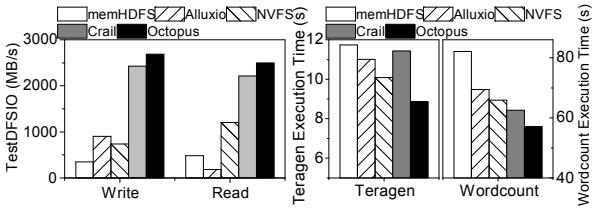


Figure 16: Big Data Evaluation

Big Data Application Performance. Figure 16(b) shows the application performance for different file systems. Octopus consumes the least time to finish all evaluated applications. Among all the evaluated file systems, memHDFS generally has the highest run time, i.e., 11.7*s* for Teragen and 82*s* for Wordcount. For the Teragen workload, the run time in Alluxio, NVFS, Crail and Octopus is 11.0*s*, 10.0*s*, 11.4*s* and 8.8*s*, respectively.

For the Wordcount workload, the run time in Alluxio, NVFS, Crail and Octopus is 69.5*s*, 65.9*s*, 62.5*s* and 57.1*s*, respectively. We conclude that our proposed general-purpose Octopus can even provide better performance for big data applications than existing dedicated file systems.

5 Related Work

Persistent Memory File Systems: In addition to file systems that are built for flash memory [17, 28, 27, 22, 44], a number of local file systems have been built from scratch to exploit both byte-addressability and persistence benefits of non-volatile memory [11, 14, 42, 32, 43]. BPFS [11] is a file system for persistent memory that directly manages non-volatile memory in a tree structure, and provides atomic data persistence using short-circuit shadow paging. PMFS [14] proposed by Intel also enables direct persistent memory access from applications by removing file system page cache with memory mapped IO. Similar to BPFS and PMFS, SCMFS [42] is a file system for persistent memory which leverages the virtual memory management of the operating system. Fine-grained management is further studied in recent NOVA [43] and HiNFS [32] to make software more efficient. The Linux kernel community also starts to support persistent memory by introducing DAX (Direct Access) to existing file systems, e.g., EXT4-DAX [4]. The efficient software design concept in these local file systems, including removing duplicated memory copies, is further studied in Octopus distributed file system to make remote accesses more efficient.

General RDMA Optimizations: RDMA provides high performance but requires careful tuning. Recent study [19] offers guidelines on how to use RDMA verbs efficiently from a low-level perspective such as in PCIe and NIC. Cell [30] dynamically balances CPU consumption and network overhead using RDMA primitives in a distributed B-tree store. PASTE [15] proposes direct NIC DMA to persistent memory to avoid data copies, for a joint optimization between network and data stores. FaSST [20] proposes to use UD (Unreliable Datagram) for RPC implementation when using `send/recv`, in order to improve scalability. RDMA has also been used to optimize distributed protocols, like shared memory access [13], replication [45], in-memory transaction [41], and lock mechanism [31]. RDMA optimizations have brought benefits to computer systems, and this motivates us to start rethinking the file system design with RDMA.

RDMA Optimizations in Key-Value Stores: RDMA features have been adopted in several key-value stores to improve performance [29, 18, 13, 40]. MICA [24] bypasses the kernel and uses a lightweight networking stack to improve data access performance in key-value stores. Pilaf [29] optimizes the *get* operation using multiple RDMA `read` commands at the client side, which offloads hash calculation burden from remote servers to clients, improving system performance. HERD [18] implements both *get* and *put* operations using the combination of RDMA `write` and UD `send`, in order to achieve high throughput. HydraDB [40] is a versatile *key-value* middleware that achieves data replication to guarantee fault-tolerance and awareness for NUMA architecture, and adds client-side cache to accelerate the *get* operation. While RDMA techniques lead to evolutions in the designs of key-value stores, its impact on file system designs is still under-exploited.

RDMA Optimizations in Distributed File Systems: Existing distributed file systems have tried to support RDMA network by substituting their communication modules [1, 3, 6]. Ceph over Accelio [3] is a project under development to support RDMA in Ceph. Accelio [2] is an RDMA-based asynchronous messaging and RPC middleware designed to improve message performance and CPU parallelism. Alluxio [23] in Spark (formerly named Tachyon) is transplanted to run on top of RDMA by Mellanox [6]. It faces the same problem as Ceph on RDMA. NVFS [16] is an optimized version of HDFS that combines both NVM and RDMA technologies. Due to heavy software design in HDFS, NVFS hardly exploits the high performance of NVM and RDMA. Crail [9] is a recently developed distributed file system built on DaRPC [37]. DaRPC is an RDMA-based RPC that tightly integrates the RPC message processing and network processing, which provides both high throughput and low latency. However, their internal file system mechanisms remain the same. In comparison, our proposed Octopus revisits the file system mechanisms with RDMA features, instead of introducing RDMA only to the communication module.

6 Conclusion

The efficiency of the file system design becomes an important design issue for storage systems that are equipped with high-speed NVM and RDMA hardware. Both the two emerging hardware technologies not only improve hardware performance, but also push back the software evolution. In this paper, we propose a distributed memory file system, Octopus, which has its internal file system mechanisms closely coupled with RDMA features. Octopus simplifies the data management layer by reducing memory copies, and rebalances network and server loads with active I/Os in clients. It also redesigns the metadata RPC and the distributed transaction by using RDMA primitives. Evaluations show that Octopus effectively explores hardware benefits, and significantly outperforms existing distributed file systems.

Acknowledgments

We thank our shepherd Michio Honda and anonymous reviewers for their feedbacks and suggestions. We also thank Weijian Xu for his contribution in the early prototype of Octopus. This work is supported by the National Natural Science Foundation of China (Grant No. 61502266, 61433008, 61232003), the Beijing Municipal Science and Technology Commission of China (Grant No. D151100000815003), and the China Postdoctoral Science Foundation (Grant No. 2016T90094, 2015M580098). Youyou Lu is also supported by the Young Elite Scientists Sponsorship Program of China Association for Science and Technology (CAST).

References

[1] GlusterFS on RDMA. "https://gluster.readthedocs.io/en/latest/AdministratorGuide/RDMATransport/".

[2] Accelio. "http://www.accelio.org", 2013.

[3] Ceph over Accelio. "https://www.cohortfs.com/ceph-over-accelio", 2014.

[4] Support ext4 on NV-DIMMs. "https://lwn.net/Articles/588218", 2014.

[5] Supporting filesystems in persistent memory. "https://lwn.net/Articles/610174", 2014.

[6] Alluxio on RDMA. "https://community.mellanox.com/docs/DOC-2128", 2015.

[7] Introducing Intel Optane technology - bringing 3D XPoint memory to storage and memory products. "https://newsroom.intel.com/press-kits/introducing-intel-optane-technology-bringing-3d-xpoint-memory-to-storage-and-memory-products/", 2016.

[8] SAP HANA, in-memory computing and real time analytics. "http://go.sap.com/product/technology-platform/hana.html", 2016.

[9] Crail: A Fast Multi-tiered Distributed Direct Access File System. https://github.com/zrlio/crail, 2017.

[10] ASSOCIATION, I. T., ET AL. *InfiniBand Architecture Specification: Release 1.3*. InfiniBand Trade Association, 2009.

[11] 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 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (New York, NY, USA, 2009), ACM, pp. 133–146.

[12] DOUGLAS, C. RDMA with PMEM: software mechanisms for enabling access to remote persistent memory. http://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/ChetDouglas_RDMA_with_PM.pdf, 2015.

[13] 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)* (2014), pp. 401–414.

[14] 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 (EuroSys)* (New York, NY, USA, 2014), ACM, pp. 15:1–15:15.

[15] HONDA, M., EGGERT, L., AND SANTRY, D. Paste: Network stacks must integrate with nvmm abstractions. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks* (2016), ACM, pp. 183–189.

[16] ISLAM, N. S., WASI-UR RAHMAN, M., LU, X., AND PANDA, D. K. High performance design for hdfs with byte-addressability of nvm and rdma. In *Proceedings of the 2016 International Conference on Supercomputing* (2016), ACM, p. 8.

[17] JOSEPHSON, W. K., BONGO, L. A., FLYNN, D., AND LI, K. DFS: A file system for virtualized flash storage. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST)* (Berkeley, CA, 2010), USENIX.

[18] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using rdma efficiently for key-value services. In *SIGCOMM* (2014).

[19] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Design guidelines for high performance rdma systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (2016).

[20] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Fasst: fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), USENIX Association, pp. 185–201.

[21] LEE, B. C., IPEK, E., MUTLU, O., AND BURGER, D. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th annual International Symposium on Computer Architecture (ISCA)* (New York, NY, USA, 2009), ACM, pp. 2–13.

[22] LEE, C., SIM, D., HWANG, J., AND CHO, S. F2FS: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)* (Santa Clara, CA, Feb. 2015), USENIX.

[23] 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).

[24] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. Mica: A holistic approach to fast in-memory key-value storage. *management 15*, 32 (2014), 36.

[25] LU, Y., SHU, J., AND SUN, L. Blurred persistence in transactional persistent memory. In *Proceedings of the 31st Conference on Massive Storage Systems and Technologies (MSST)* (2015), IEEE, pp. 1–13.

[26] LU, Y., SHU, J., SUN, L., AND MUTLU, O. Loose-ordering consistency for persistent memory. In *Proceedings of the IEEE 32nd International Conference on Computer Design (ICCD)* (2014), IEEE.

[27] LU, Y., SHU, J., AND WANG, W. ReconFS: A reconstructable file system on flash storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)* (Berkeley, CA, 2014), USENIX, pp. 75–88.

[28] LU, Y., SHU, J., AND ZHENG, W. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)* (Berkeley, CA, 2013), USENIX.

[29] MITCHELL, C., GENG, Y., AND LI, J. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)* (2013), pp. 103–114.

[30] 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)* (2016).

[31] NARRAVULA, S., MARNIDALA, A., VISHNU, A., VAIDYANATHAN, K., AND PANDA, D. K. High performance distributed lock management services using network-based remote atomic operations. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid'07)* (2007), IEEE, pp. 583–590.

[32] OU, J., SHU, J., AND LU, Y. A high performance file system for non-volatile main memory. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 12.

[33] PELLEY, S., CHEN, P. M., AND WENISCH, T. F. Memory persistency. In *Proceedings of the 41st ACM/IEEE International Symposium on Computer Architecture (ISCA)* (2014), pp. 265–276.

[34] QURESHI, M. K., SRINIVASAN, V., AND RIVERS, J. A. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual International Symposium on Computer Architecture (ISCA)* (New York, NY, USA, 2009), ACM, pp. 24–33.

[35] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The hadoop distributed file system. In *IEEE 26th symposium on mass storage systems and technologies (MSST)* (2010), IEEE, pp. 1–10.

[36] STRUKOV, D. B., SNIDER, G. S., STEWART, D. R., AND WILLIAMS, R. S. The missing memristor found. *nature 453*, 7191 (2008), 80–83.

[37] STUEDI, P., TRIVEDI, A., METZLER, B., AND PFEFFERLE, J. DaRPC: Data center rpc. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)* (2014), ACM, pp. 1–13.

[38] SWANSON, S., AND CAULFIELD, A. M. Refactor, reduce, recycle: Restructuring the i/o stack for the future of storage. *Computer 46*, 8 (2013), 52–59.

[39] TALPEY, T. Remote Access to ultra-low-latency storage. `http://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/Talpey-Remote_Access_Storage.pdf`, 2015.

[40] 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* (2015), ACM, p. 22.

[41] 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* (2015), ACM, pp. 87–104.

[42] WU, X., AND REDDY, A. L. N. SCMFS: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (New York, NY, USA, 2011), ACM, pp. 39:1–39:11.

[43] XU, J., AND SWANSON, S. Nova: a log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (2016), pp. 323–338.

[44] ZHANG, J., SHU, J., AND LU, Y. Parafs: A log-structured file system to exploit the internal parallelism of flash devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (2016).

[45] 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.

[46] ZHOU, P., ZHAO, B., YANG, J., AND ZHANG, Y. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th annual International Symposium on Computer Architecture (ISCA)* (New York, NY, USA, 2009), ACM, pp. 14–23.

iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call

Daejun Park and Dongkun Shin
Sungkyunkwan University, Korea
pdaejun@skku.edu, dongkun@skku.edu

Abstract

For data durability, many applications rely on synchronous operations such as an `fsync()` system call. However, latency-sensitive synchronous operations can be delayed under the compound transaction scheme of the current journaling technique. Because a compound transaction includes irrelevant data and metadata, as well as the data and metadata of fsynced file, the latency of an fsync call can be unexpectedly long. In this paper, we first analyze various factors that may delay an fsync operation, and propose a novel hybrid journaling technique, called `ijournaling`, which journals only the corresponding file-level transaction for an fsync call, while recording a normal journal transaction during periodic journaling. The file-level transaction journal has only the related metadata updates of the fsynced file. By removing several factors detrimental to fsync latency, the proposed technique can reduce the fsync latency, mitigate the interference between fsync-intensive threads, and provide high manycore scalability. Experiments using a smartphone and a desktop computer showed significant improvements in fsync latency through the use of `ijournaling`.

1 Introduction

The buffered I/O is essential to a high-performance file system because data can be temporarily buffered in the main memory until being written back to storage. However, the buffered I/O cannot guarantee file-system consistency and data durability in the cases of unclean file-system shutdowns or hardware failures [22]. To ensure file-system consistency, many file systems have adopted a journaling technique, which can ensure the atomicity of a transaction. A transaction is a group of file system modifications that must be carried out atomically. For example, the `ext4` file system uses the journaling block device version 2 (JBD2) in the Linux kernel to support journaling [23]. All file system operations are logged in the journal area before updating the original file system. Therefore, by undoing any incomplete transactions and redoing all committed transactions, journaling can be used to maintain the file-system consistency despite sudden system crashes.

The `ext4` file system uses a physical logging scheme that records the modified blocks [14], rather than logical logs, which records operations. Because several of the metadata structures of `ext4`, such as `block bitmap` and `inode table`, are shared among multiple file operations, it is easier and more efficient to commit multiple transactions at once rather than commit each file operation-level transaction individually. For the purpose, `ext4` groups concurrent unrelated transactions into a single compound transaction [26], which is periodically flushed into a reserved storage area, called a journal area. The compound transactions are maintained in the journal transaction buffer of the main memory until being committed to the journal area. The compound transaction scheme provides a better performance, particularly when the same metadata structure is frequently updated within a short period of time.

`Ext4` supports three journaling modes: writeback, ordered, and data modes. Ordered mode, which is the default option, journals only the metadata. However, it enforces an ordering constraint to guarantee file-system consistency, in which the transaction-related data writes must be completed before the journal writes of the metadata. Therefore, the transaction commit latency will be lengthy if the size of the associated data is large. A long transaction commit latency may not be a serious problem, however, because the journal commit operations are periodically invoked by a background journaling thread.

Although the file-system consistency and data durability are supported using a journaling scheme, the data durability is not immediate. To ensure instant data durability, users must call a synchronous operation such as an `fsync()` or `fdatasync()`. Most database systems rely

on fsync system calls to be assured of immediate data durability. Recent mobile platforms such as Android also frequently use fsync system calls [17]. Because an fsync system call is a synchronous operation, the fsync latency affects the performance of the application.

When a file system uses journaling, all file-system changes are updated through the journaling layer. Therefore, when an application calls an fsync for a modified file, the journaling thread is awakened on demand, and the transactions in the transaction buffer are flushed into the storage immediately, irrespective of the journal commit interval. The fsync operation must wait until the journal commit operation is completed. In particular, a compound transaction in the transaction buffer may include data and metadata updates of other irrelevant files, as well as the target file of the fsync call (*fsynced* file). A long latency for committing a compound journal transaction will increase the latency of an fsync system call [12].

For a short fsync latency, a more fine-grained journaling scheme such as file-level transaction committing is required. However, under a physical logging scheme, fine-grained journaling is difficult to implement because several metadata blocks are shared by multiple file operations. In addition, fine-grained journaling imposes a high journaling overhead.

Another solution is the use of a logical logging scheme. For example, `XFS` [28] and `ZFS` [7] log the file operations rather the modified blocks for a synchronous request. All file system operations are logically-logged as transactions, which accumulate in memory until they are committed to the journal area for an fsync call. The logical logs are replayed during a crash recovery. However, logical logging requires a large sized transaction buffer in the memory compared with physical logging, particularly when the same metadata structure is frequently updated. For example, ZFS generates a 256 bytes of logical log in memory for each write operation.

To address this issue, we propose a hybrid approach that uses both the normal journaling by JBD2 and the file-level transaction journaling of our proposed `ijournaling` technique. Under a normal periodic journaling operation, the proposed scheme uses a legacy journaling scheme that flushes the compound transaction. However, if on-demand journaling is invoked by an fsync call, `ijournaling` commits only the transactions related to the fsynced file without flushing the compound transaction in the transaction buffer. The file-level transactions include only the minimum metadata, through which all relevant file-system metadata blocks can be recovered after a system crash. The `ijournaling` technique can eliminate the compound transaction problem for an fsync call without requiring an additional large amount of memory space for transaction management, unlike `ZFS`. We evaluated the performance improvements of the proposed journaling scheme on both a smartphone and a desktop system.

2 Background

`Ext4` is the default file system of Linux kernel, and is widely used on mobile devices such as Android-based smartphones and desktop computers. `Ext4` divides an entire storage space into several block groups. Two metadata structures, i.e., `superblock` and `group descriptor table` (`GDT`), describe the general information of the overall file system. Each block group has its own `block bitmap` and `inode bitmap` to manage the allocation status of the data blocks or inode entries. Each block group also maintains an `inode table`. Each inode entry of the `inode table` is 256 bytes in size and describes the attributes of a single file or directory. These metadata structures are allocated in a 4-KB block unit, and are shared by multiple files or directories. `Ext4` supports an extent-based block-mapping scheme. A single extent identifies a set of blocks that are logically contiguous within the file and also on the underlying block device. An inode entry can contain a maximum of four extent structures internally. If more extents are required, external extent structures are allocated in the data block area for indirect pointing.

`Ext4` uses a journaling technique. Information regarding pending file-system updates is first written to the journal to enable an efficient crash recovery. The journal space is treated as a circular buffer. Once the necessary information has been propagated to its fixed location in the `ext4` structures, the corresponding journal logs are identified as *checkpointed*, and the space can be reclaimed. All modified metadata blocks are recorded in a block unit at the journal area even though only a portion of the metadata blocks is modified. This feature makes it difficult to implement file-level journaling because a metadata block is shared by multiple files. One transaction log in the journal contains a journal header (JH), several journal descriptor blocks (JDs) to describe its contents, and a journal commit block (JC) to denote the end of the transaction.

`Ext4` manages the life cycle of each transaction. Each transaction has a metadata list and an inode list, which have the metadata blocks and pointers to the inodes modified by the transaction, respectively. First, a *running* transaction is created, and all file-system modifications are inserted into the running transaction. When the periodic JBD2 thread is invoked or an `fsync()` is called, the transaction state is changed to *committing*, and the transaction blocks are written into the journal area. After the completion of a transaction commit, the transaction is marked as *checkpoint*. After the transaction is checkpointed, it is removed from the transaction list.

3 Related Work

Prabhakaran *et al.* [26] observed the storage performance when a foreground asynchronous sequential stream and a background random synchronous stream compete to use the `ext3` file system. They showed that the more frequently the background process calls an fsync, the more traffic is sent to the journal owing to the compound transactions of `ext3`. The authors proposed an adaptive approach that selects the best journaling mode for each transaction according to its I/O pattern. However, this approach cannot solve the compound transaction problem completely, and may be unsafe [27].

Jeong *et al.* [17] revealed the journaling-of-journal (JoJ) problem on an Android-based smartphone, where the `ext4` file system uses a journaling scheme for data reliability, and SQLite [3] conducts additional journaling using its own journal file. Their study suggests using `fdatasync()` and write-ahead logging (WAL) in SQLite to reduce the number of journal commits. Here, `fdatasync()` does not commit a journal transaction unless the file-system metadata relevant to the target file are changed. However, WAL also generates frequent fsync calls, and `fdatasync()` can be effective only when there are no metadata updates.

To mitigate the JoJ overhead, Shen *et al.* [27] proposed using the data journaling mode of `ext4` adaptively. Data journaling writes both data and metadata in the journal area without generating page writes at the original file system locations during a journal commit operation. Because a journal commit operation sends only the sequential write requests to the storage, the journal commit latency can be reduced. However, this technique also flushes compound transactions and cannot completely avoid a long fsync latency.

There are several approaches that divide a file system space into several groups to localize the faults and transactions of the filesystem, or to avoid the lock contention on shared file-system data structures in memory. The per-block-group (PBG) journaling scheme [19] exploits the block groups of `ext4`. Because each block group has its own metadata blocks, PBG journaling extracts a block-group-level transaction including updates on the fsynced file from a compound transaction, and commits only the transaction of the target block group. PBG journaling shows significant improvements in terms of fsync latency when a fsynced file and other irrelevant files are allocated in different block groups. However, a long fsync latency occurs if irrelevant files share the same block group. The eager synching [8] also uses a similar technique as PBG journaling.

IceFS [21] proposed a new container abstraction, called *cube*, to provide more flexible and configurable isolations. SpanFS [18] distributes files and directories among the *domains*, which are the basic independent function units for file system services such as data allocation and journaling. IceFS and SpanFS also cannot avoid the compound transaction problem within a cube or domain. Moreover, IceFS is incompatible with legacy file systems, and the user should manage the cubes. SpanFS can generate a large compound transaction across multiple domains. Xsyncfs [25], NoFS [10], and OptFS [9] improved the fsync latency by delaying sync operations or changing the implementation of ordering constraint.

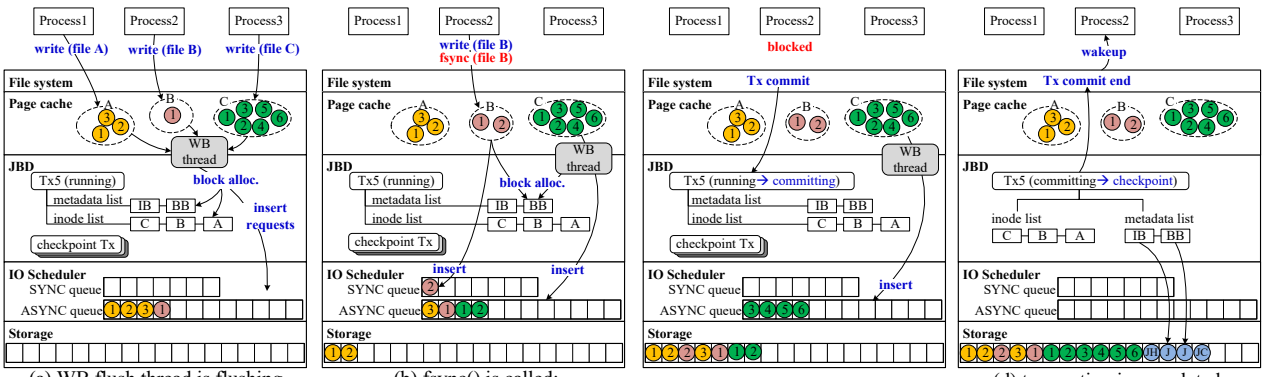
ScaleFS [13] uses a logical logging technique. Operation logs (OpLogs) are generated in its in-memory file system to record file-system changes. An OpLog consists of logical file-system operations, and is applied to the on-disk file system when an fsync is invoked. ScaleFS applies only dependent operations that are related to the file or directory being fsynced, which is a very similar approach to our proposed `ijournaling` technique. However, logical logging-based journaling scheme requires significant changes to the current `ext4` file systems. In addition, a performance overhead occurs because each file-system operation must record its own OpLog. Our proposed `ijournaling` follows the physical logging scheme of `ext4`, and has little overhead for managing file-level journals.

Jeong *et al.* [15] proposed an I/O scheduler technique that can detect asynchronous I/O requests related with latency-sensitive file operations such as an fsync call, and boost them over the other asynchronous I/Os. This technique improves the fsync latency and can be used along with our technique because they both handle the different underlying reasons for a long fsync latency problem. However, the number of latency-sensitive asynchronous I/Os can be minimized under our `ijournaling` scheme because only the relevant blocks are flushed by fsync calls.

Min *et al.* [24] investigated the performance of `fsync()` for a manycore architecture under five widely-deployed file systems. They showed that most of the file systems start to degrade in performance when more than ten cores compete for the file system. In our `ijournaling` scheme, a sync operation does not depend on a single journaling thread and each core has its own separate ijournal area. Therefore, our scheme provides a better manycore scalability, which is described in greater detail in Section 6.

4 Analysis of Fsync Latency in `Ext4`

When a user process calls an `fsync()` system call for a file, the process is blocked, and the system call service in the kernel performs the following operations, as shown in Figure 1. First, it updates the related metadata blocks for the file, inserts them into the running transaction man-



(a) WB flush thread is flushing dirty pages

(b) fsync() is called; data pages are flushed

(c) commit transaction

(d) transaction is completed; fsync() is completed

Figure 1: Dependency problems of a journal commit.

aged by JBD2, and flushes the data blocks of the fsynced file, as shown in Figure 1(b). For example, the `block bitmap` needs to be modified when an fsync call flushes newly allocated data blocks. `Ext4` uses a delayed block allocation scheme, and thus, the file-system location for a data block is determined just before the block is flushed into storage. The write requests on the data blocks of the fsynced file are transferred as synchronous requests because the user process is waiting for the completion of the system call.

Second, the system call service sends a commit request for the relevant transaction to JBD2 if the transaction state is still running, and waits for the completion of the commit operation, as shown in Figure 1(c). In this step, a commit operation cannot be issued immediately if there is another committing transaction because JBD2 can commit only one transaction at a time. During a commit operation, JBD2 awaits the completion of all data write requests relevant to the target transaction. In Figure 1(c), all data blocks of files **A**, **B**, and **C** must be flushed because the target transaction includes the inodes. Finally, JBD2 writes the journal blocks in the journal area after the completion of the data write operations, as shown in Figure 1(d). A journal block includes the modified metadata blocks. The final block written by JBD2 is the journal commit (JC) block, which is followed by a *flush* command. When the *flush* command is completed, the `fsync()` system call is completed, and the user process can continue with its operations.

Based on its operations, we can find several reasons for adverse effect on the latency of an fsync system call. The first reason is the inter-transaction (IT) dependency. Because `ext4` uses a single JBD2 thread, only one transaction (i.e., a committing transaction) can be committed at a time. Protecting concurrent journal commits is important for preventing multiple journals from being interleaved in the journal area. Furthermore, multiple transactions cannot be committed concurrently because they share several metadata blocks. Therefore, if the JBD2 thread is committing transaction Tx_{n-1}, the next transaction Tx_n relevant to the fsynced file cannot be changed into a committing transaction immediately. Such cases will occur frequently when multiple threads invoke fsync calls simultaneously. To solve this IT dependency problem, our `ijournaling` technique handles an fsync call at system call service rather than the journaling thread, and uses separated journal areas.

The second reason is the compound transaction (CTX) dependency, shown in Figure 1(c). When the JBD2 thread commits the transaction of an fsynced file, the inode list of the committing transaction includes irrelevant inodes. The JBD2 thread must wait for the completion of the data block write operations owing to the ordering constraint of ordered-mode journaling. The CTX dependency is severe when there are many processes generating file-system write operations. Even when only one process generates write operations, a CTX dependency problem can occur if the process updates multiple files. In some cases, a transaction can include *discard* commands [1], which have considerably long latencies.

The delayed block allocation technique of `ext4` aggravates the CTX problem. The delayed block allocation has many advantages because it postpones block allocations until the page flush time, rather than during a `write()` operation [23]. Therefore, the overall performance of the file system is higher when delayed allocation is enabled. However, if an fsync is called just after the flush kernel thread invocation, as shown in the example in Figure 1(a), the flush thread will allocate data blocks for dirty pages, and register several modified inodes in the running transaction during the delayed block allocation. Then, the commit operation of the journal transaction will generate many write requests into storage. If an fsync is called before the flush thread is invoked, the fsync latency will be short because there are few modifications to the file system. Therefore, fsync latencies will fluctuate in a delayed allocation scheme. On the contrary, if the delayed allocation is disabled, the

modified inodes will be distributed to different transactions, and the fsync latency will be unrelated with the flush thread invocation. Nevertheless, a delayed allocation can demonstrate a better performance and shorter average fsync latency, as described later in Section 6. Because our `ijournaling` scheme commits a file-level transaction rather than a compound transaction, it can always demonstrate a short fsync latency irrespective of the block allocation policy. Throughout our study, we used delayed allocation as the default scheme.

The last reason is the quasi-async request (QA) dependency revealed in [15]. In Figure 1(a), the writeback flush thread has sent a write request on data block 1 of file **B** before an fsync is called. Whereas the write requests generated by an fsync system call are sent along with a `SYNC` flag, the write requests generated by the flush thread are sent without the flag. The CFQ I/O scheduler in Linux gives lower priorities to requests without a `SYNC` flag. Although data block 1 is written by an async request, the request is latency-sensitive. Such a request is called a *quasi-async* request. A long latency will occur for completion of the quasi-async request, particularly when there are many competing async requests in the I/O queue. The QA dependency problem can be solved through the boosting technique proposed in [15], which changes a quasi-async request into a sync request. However, owing to the CTX dependency, the asynchronous write requests on **A** and **C** in Figure 1 must also be changed to sync requests in the boosting technique. The `ijournaling` can mitigate the QA dependency problem by removing unrelated dependencies. For example, the fsync call on **B** does not need to wait for the completion of write requests on **A** and **C**.

5 The iJournaling Scheme

5.1 Main Idea

The goal of `ijournaling` is to improve the performance of an `fsync()` call while exploiting the advantage of the legacy compound-transaction-based journaling scheme. Only when a process calls an `fsync()` system call, `ijournaling` is invoked. The `ijournaling` scheme generates `ijournal` transactions (i-transactions) and flushes them into a reserved `ijournal` area without committing the normal running transaction of an fsynced file. The i-transaction includes metadata modification logs, which are the minimum required information through which a crash recovery operation can recover the file-system metadata blocks modified through an fsync operation. Only file-level metadata such as an inode entry and the external extent structures of the target file, and any related directory entries (DEs), are recorded. Other modified metadata blocks shared by other files, such as `GDT`, `block bitmap`, `inode bitmap`, or `inode table`, are not flushed into the `ijournal` area. They can be recovered during the crash recovery time using committed i-transactions. The `ijournaling` scheme does not change the normal running transaction used by the JBD2 thread. Therefore, the metadata blocks committed by `ijournaling` are again committed into the normal journal area through the following periodic JBD2 thread, which simplifies the crash recovery.

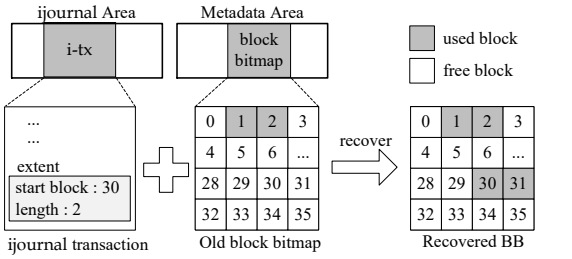


Figure 2: `Block bitmap` recovery with ijournal.

Figure 2 shows an example of a metadata recovery operation of `ijournaling`. When the file-system recovery module finds a committed i-transaction in the `ijournal` area, it can modify the old `block bitmap` in the file system using the extent allocation information, which can be found from the inode entry or the external extent structures in the i-transaction. Because two blocks from block number 30 are allocated for an extent, the 30-*th* and 31-*st* bits in the `block bitmap` must be set. The `inode table` and `inode bitmap` can also be easily recovered through a recorded inode entry. To implement `ijournaling`, no changes are required to the current JBD2 journaling scheme. Whereas a normal journaling thread flushes the transaction buffer periodically, `ijournaling` is performed in the `fsync()` system call service. Therefore, an `ijournaling` and a normal journaling can be performed simultaneously, and the inter-transaction dependency is removed. The file-system recovery module must be modified to handle `ijournal`.

5.2 iJournal Transaction

The `ijournal` area is separated from the normal journal area. In addition, each processor core uses a separate per-core `ijournal` area in order to support manycore scalability. Each `ijournal` area is managed as a circular buffer. This scheme needs to allocate space as many as the number of cores. If the existing normal journal area is shared by normal journal transactions and i-transactions, no additional space allocation is required. However, we should be carefully in allocating blocks in the journal area to prevent two different journal blocks from being mixed in the journal area in an interleaved manner, because a transaction must consist of consecutive blocks.

While a JBD2 thread is allocating blocks in the journal area, the `ijournaling` must wait until the block allocation is completed. Therefore, separating journal areas can improve the concurrency of journaling operations. The required storage space for per-core `ijournal` area is small because the the size of an i-transaction is smaller than that of a normal transaction, and i-transactions will be invalidated after its corresponding normal transaction is committed.

Figure 3 shows the structure of an i-transaction, of which there are two types: file i-transaction and directory i-transaction. Whereas the file i-transaction has the metadata information of an fsynced file, the directory i-transaction has the metadata information of any related parent directory.

A file i-transaction is composed of one header block, several external extent blocks (if they exist), and one commit block. The journal header in the header block has the same structure as a normal journal header. It includes the magic number and transaction ID. A file i-transaction has the same transaction ID as the running transaction of normal journaling, which includes the metadata updates of the corresponding fsynced file. Because the journal transactions are distributed among multiple journal areas, the crash recovery module must identify the order of each transaction based on its transaction IDs. Because there can be multiple fsync calls before the current running transaction of normal journaling is committed, several i-transactions will have the same transaction IDs. In particular, for the i-transactions recorded at different `ijournal` areas, it is impossible to know the order of them if they have a same transaction ID. To resolve this problem, `ijournaling` uses a sub-transaction ID, which is incremented by each fsync call and managed globally among multiple cores.

The inode number and inode structure in an i-transaction are used for recovering the `inode table`, `inode bitmap`, and `GDT`. Each block tag stores the mapping between an external extent block in the file i-transaction and its actual file-system block number. The crash recovery can update the `block bitmap` using the internal extent information in the inode structure, the block tags, and the external extent blocks. The file i-transaction collects only dirty external extent structures. To reduce the extent tree search overhead, we modified the file system to maintain a list of dirty extent blocks for each uncommitted file and update it during each extent allocation/free operation. Because only a 20 bytes of data structure is required for tracking one external extent, the memory overhead for external extent tracking is not significant. The commit block indicates whether an i-transaction has been completely committed.

The directory i-transaction is used to record any relevant directory updates. If a file is fsynced but its parent

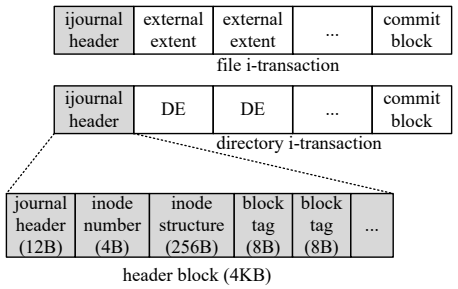


Figure 3: Structure of ijournal transaction.

directory entry is not committed before a system crash, the file will be unreachable after the system recovery. For example, if directory `A` and its subdirectory `B` are created, and an fsync call for file `/A/B/c` is called, `ijournaling` records all the changed directory information of the directories of `A` and `B`, as well as the changed file information of file `c`. The `ijournaling` identifies all directories that are related to the fsynced file and therefore must also be committed.

To track the uncommitted directories, we added the `uncommitted_DE` flag in the inode structure. When a new file is created, the flag is marked in the created file's inode to denote that its directory entry has not been recorded at the parent directory block. The flag is cleared when the parent directory block is committed by JBD2. The `ijournaling` first checks the flag of the fsynced inode. If the flag is marked, the parent directory is also examined recursively until no more uncommitted directory is found. At that time, the directory i-transaction of topmost uncommitted parent directory is first written in the `ijournal` area, and then the directory i-transactions of next-level directories are written in order. Finally, the file i-transaction of fsynced file is written. Therefore, even though there is a system crash during the ijournaling, the recovered file system can maintain its consistency (i.e., there is no unreachable file or directory.) Although only one directory entry in the DE blocks of an uncommitted directory is related with an fsync call, our scheme records the entire DE blocks of the uncommitted directory in the directory i-transaction for fast fsync handling, because it is time consuming to extract the modified directory entries from the DE blocks. Instead, the recovery process identifies the modified and valid DE entries to update the old DE blocks in the file system.

If there are no modified external extent blocks and DE blocks to be committed by an fsync call, it will be possible to write a single block i-transaction by recording all information in the ijournal header, which can reduce the write traffic on the ijournal area.

The ijournaling will show a slightly difference on crash recovery compared with the normal journaling scheme. While the normal journaling can recover all the other contemporary file operations as well as the fsynced

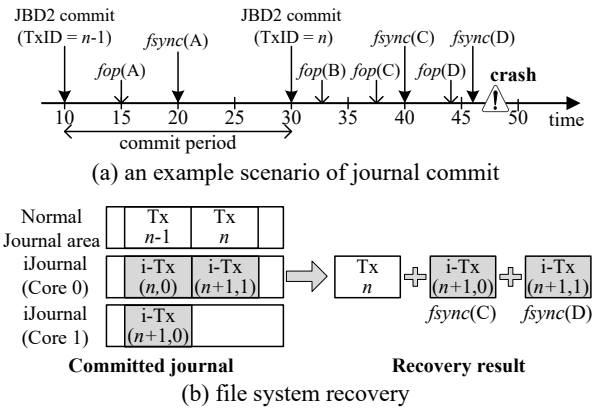
file operation, the proposed ijournaling can recover only the files and directories related to fsync operation. However, the file system consistency is guaranteed.

To simplify the `ijournaling` implementation, our scheme uses the normal journaling for some cases. For the fsync call for a directory itself, a normal transaction is committed instead of an ijournal to record all file-system changes in the subdirectories, as well as in the fsynced directory entry. This simplifies the journaling by removing the traversing of the subdirectories. When an inode is shared by multiple files using hard link and an `fsync()` is called for only one file, the file-system consistency can be broken if `ijournaling` records the parent directories of only the fsynced file. To eliminate the traversing of directories connected by hard links, a normal transaction is committed instead of an ijournal for the case. To track such a case, we added the `uncommitted_HL` flag in the inode structure. The flag of a file is marked if the `i_link_count` of its inode is incremented by a hard link operation. The flag is cleared when a running transaction is committed by the JBD2 thread. The fsync system call service checks the flag of the target inode, and calls normal journaling if the flag has been marked.

5.3 Crash Recovery

The ijournal crash recovery module replays only valid i-transactions. It first scans the normal journal area, replays the committed but not-yet-checkpointed journal transactions, and finds the last committed journal transaction ID (`Max_TxID`). Because valid i-transactions have the information on file-system changes after a valid normal journal transaction is committed, the normal journal transaction must be replayed before i-transactions. Then, the recovery module scans the `ijournal` areas. If an i-transaction has a transaction ID larger than `Max_TxID`, it is valid. Otherwise, the i-transaction is ignored since a normal committed journal transaction includes all the metadata modifications of the i-transaction. If there are multiple i-transactions on an inode, only the last i-transaction with the largest sub-transaction ID is valid since the last one includes all the metadata modifications of the previous i-transactions.

Figure 4(a) shows an example of journal commit. At a time of 30, the normal transaction with the transaction ID (TxID) n is committed and the TxID is incremented to $n+1$. Before the next periodic transaction with TxID = $n+1$ is committed, the files **B**, **C**, and **D** are modified, and `fsync()` calls are invoked for the files **C** and **D** by different processor cores. In Figure 4(b), the i-transactions with (TxID, sub-TxID) = $(n+1,0)$ and $(n+1,1)$ have the committed file information of the files **C** and **D**, respectively. The system is crashed before the periodic transaction commit (TxID = $n+1$). In Figure 4(b), the i-transaction with TxID = n is invalid because the normal transaction with TxID = n has been committed. Therefore, the recovery operation uses only the i-transactions with TxID = $n+1$. In Figure 4(a), there is a file operation on file **B** before a system crash, but the operation cannot be recovered by `ijournaling`. However, there is no problem in file-system consistency.



(a) an example scenario of journal commit

(b) file system recovery

Figure 4: Example of journal commit and recovery.

For each valid i-transaction, the recovery module modifies the corresponding inode entry and other metadata blocks in the file system. Because an fsync call can generate one file i-transaction and multiple directory i-transactions, the multiple i-transactions generated by an fsync call cannot be committed atomically if a system crash occurs during fsync handling. In addition, the DE blocks in directory i-transaction also contain information on irrelevant files. Instead of directly copying the DE blocks of a directory i-transaction into the file-system blocks during a crash recovery, the crash recovery operation first identifies the changed directory entries by comparing the two different DE blocks. If the inode pointed to by a changed directory entry is accessible, the entry is modified in the DE blocks in the file system.

Figure 5 shows an example of a file-system recovery under the `ijournaling` scheme. Initially, the file with inode number 3 has three external extents, which are used to access 24 blocks. Through some file operations, ten blocks (block numbers 50-59) and the corresponding external extent structure in block number 12 are freed. Then, six blocks (block numbers 74-79) are appended, and the external extent in block number 13 is modified. After the file operations, an fsync is called. Assume that there is a system crash before a normal journal is committed. The recovery module builds the inode structure including the external extent tree with the recorded i-transactions. By comparing the built inode with the corresponding inode in storage, the recovery module can identify the file-system changes by the logged fsync call, and can replay these changes. When the external extent block in block number 12 is freed, the original `ext4`

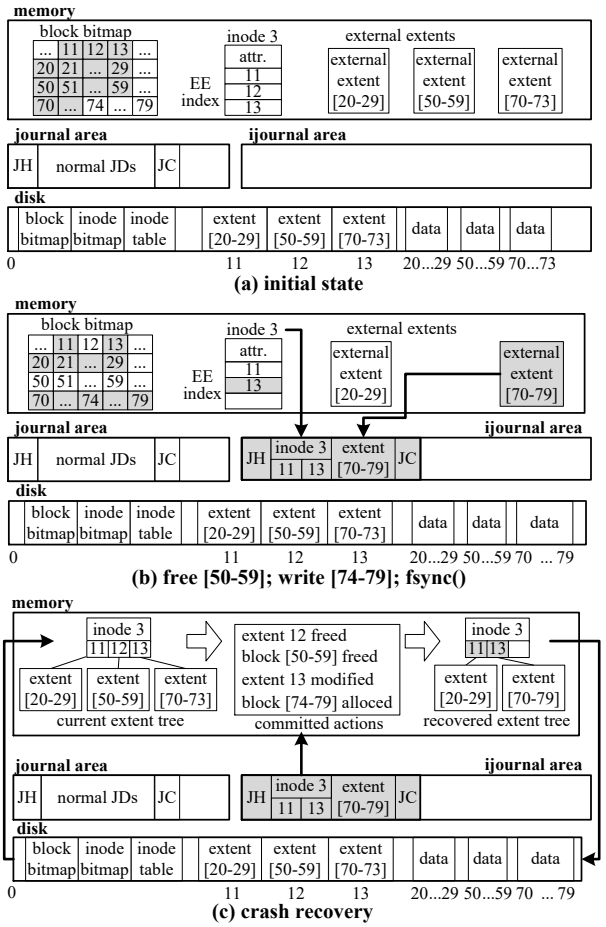


Figure 5: Example of a crash recovery.

journaling records a *revocation* block at the journal area to prevent an incorrect replay of the journal, which will cause a data corruption. The `ijournaling` scheme skips the writing of the revocation block because the following normal journaling will write it.

6 Experiments

6.1 Experiment Environments

To evaluate the effectiveness of `ijournaling`, an Android-based smartphone and a desktop computer were used. The smartphone was equipped with a Samsung Exynos 5410 (1.6-GHz Quad Cortex-A15 + 1.2-GHz Quad Cortex-A7) processor, 2 GB of DRAM, and 32 GB of eMMC. The Android OS version was 4.2.2 (Jelly Bean), and the Linux kernel version was 3.4.5. The desktop computer was equipped with an Intel i7-4790 3.6-GHz CPU, 16 GB of DRAM, and a Samsung 850 Pro SSD. The desktop Linux version was 4.7.3. The delayed allocation and ordered-mode journaling were used by default. The JBD2 thread conducts a journal commit operation at periodic 5-second intervals.

Linux kernel version 3.8 or later removes the ordering constraint of the ordered-mode journaling scheme [29]. Therefore, it is not necessary for an fsync call to wait until all data pages relevant to the journal transaction are flushed into the disk. However, the modified ordered-mode journaling scheme cannot guarantee file-system consistency similar to writeback-mode journaling. This flaw has been fixed at version 4.6.2 [20]. The Linux kernel versions used in our experiments (i.e., 3.4.5 and 4.7.3) keep a strict ordering constraint in ordered mode journaling.

6.2 Basic Comparison

We first measured the fsync latencies under different journaling schemes, normal and `ijournaling`, on the desktop and smartphone. The boosting technique [15] was optionally applied. We ran two programs for the experiments. One is an fsync-generating thread (*fsync tester*), which writes 80 KB of data in a file and calls an fsync repeatedly. We gave a delay of 0.1 second between `write()` and `fsync()` in order to generate many quasi-async requests. The other is the fio program [6], which generates 4 KB of sequential write requests for a file with a configurable write bandwidth of BG_{bw}. The fio program was used as a background process, which generated many data blocks to be flushed during the transaction commit operation. We determined the value of BG_{bw} at each experiment considering the storage bandwidth and the target foreground workload.

Figure 6(a) shows the results for the desktop when BG_{bw} = 400 MB/s. In the normal journaling scheme, the tail fsync latency at the 95th percentile is longer than 3.5 seconds. This is because the fsync must wait until a large number of dirty pages are flushed. In our measurement, 1.5 GB of data blocks at maximum were flushed during an fsync handling. However, `ijournaling` showed less than 0.2 seconds of fsync latency. The boosting technique was not very effective at reducing the fsync latency. Because SSD supports command queueing, most of the quasi-async requests were sent to storage without a long delay in the I/O scheduler. Once a request is sent to storage, the boosting cannot be applied because the host system cannot control the transferred requests.

Figure 6(b) shows the results for the smartphone when BG_{bw} = 50 MB/s. The `ijournaling` scheme also improved the fsync latency in the smartphone. Unlike with the desktop experiments, the boosting technique was effective because eMMC is slower than SSD, and does not support command queueing. By removing the CTX dependency, `ijournaling` significantly reduced the number of quasi-async requests and showed a shorter 95th percentile tail latency without boosting. We also implemented the logical logging scheme in the `ext4` file

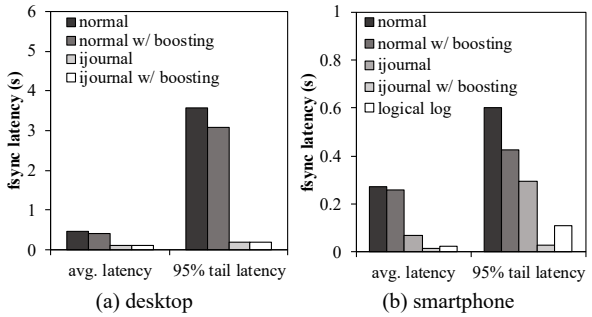


Figure 6: Fsync latency for different journaling schemes.

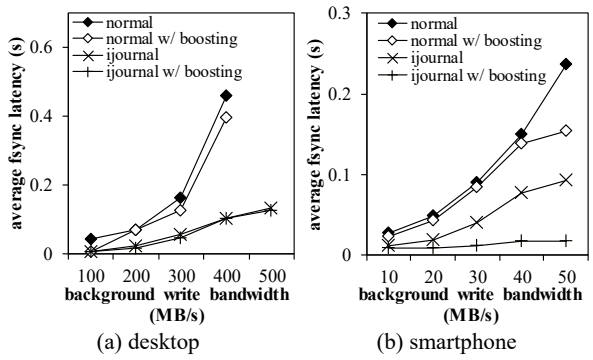


Figure 7: Changes in fsync latency when varying the number of concurrent file operations.

system. We followed the design of logical logging in ZFS. The delayed allocation was disabled in the logical logging experiments because the logical logging must generate an operation log for each file operation. The logical logging showed longer latencies compared with `ijournaling` using the boosting scheme. This is because the logical logging must flush a large size of logs.

To demonstrate the CTX dependency problem in legacy journaling, we measured the fsync latencies of *fsync tester* while varying the write bandwidth of the background process, i.e., BG_{bw} of fio. Figure 7 shows the average fsync latencies under four different journaling schemes. As the background write bandwidth increased, the fsync latency increased for the normal journaling scheme because more transactions were merged into a compound transaction. In particular, when BG_{bw} = 500 MB/s during the desktop experiment, the fsync system call was not completed until the background fio program was terminated. However, the `ijournaling` scheme showed short latencies even when BG_{bw} was high. The booting scheme was effective only when `ijournaling` is enabled.

Figure 8 compares the fsync latencies in legacy journaling under different block allocation policies. The experiment scenario is same as the scenario of Figure 6(a). When an `fsync()` was called while the flush thread was flushing dirty pages, the fsync latency became signifi-

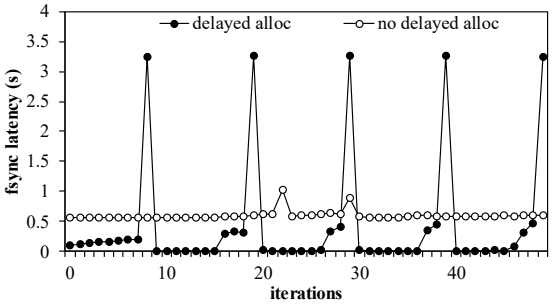


Figure 8: Fsync latency with and without delayed allocation at a desktop.

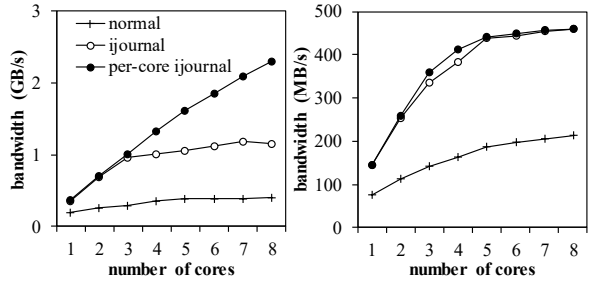
cantly high for the delayed allocation scheme. Otherwise, the latency was short. This is because data blocks are allocated when the flush thread is invoked. However, when the delayed allocation is disabled, there are no significant changes in the fsync latency. The average fsync latency is shorter when the delayed allocation is enabled. Because `ijournaling` can solve the CTX dependency problem, it can mitigate the fluctuating fsync latency problem of delayed allocation, and thus showed less than 0.2 seconds latencies as shown in Figure 6(a).

6.3 Manycore Scalability

A critical hurdle in implementing a manycore-scalable file system is the journaling contention, as reported in [24]. In particular, a single JBD2 thread handles all file-system transactions in `ext4`. Because `ijournaling` commits an fsync-related transaction in the system call service without calling the JBD2 thread, it improves the manycore scalability. In addition, each core has its own `ijournal` area, and thus, multiple fsync calls can be handled simultaneously at multiple processor cores.

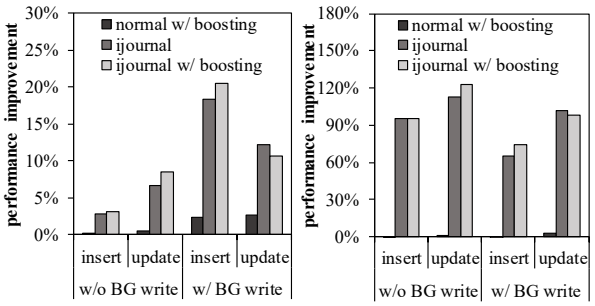
In this experiment, we used a Xeon E5-2630 machine equipped with 2.4 GHz 8-core CPU, 64 GB of DRAM, and an Intel 750 NVMe SSD (400GB). The Linux kernel version was 4.7.3. Each core ran a process of sysbench [4], which generated 4 KB of sequential write requests on 128 files. Each `write()` operation was followed by an `fsync()` call. Figure 9 shows the changes in total bandwidth of the multiple sysbench processes while increasing the number of processor cores. Three different journal schemes were tested: normal journaling, `ijournaling` with one shared ijournal area, and `ijournaling` with a separate ijournal area per core.

The rate of increase in the total bandwidth decreased in normal journaling owing to its inter-transaction dependency problem. While JBD2 commits the transaction of a process, other processes must await the completion of the transaction commit. However, `ijournaling` improves the bandwidth significantly. In particular, when a separate `ijournal` area was allocated for each core



(a) ramdisk (b) Intel 750 NVMe SSD

Figure 9: Multicore scalability.



(a) WAL journal (b) rollback journal

Figure 10: Mobibench results on a smartphone.

and the storage device used was a ramdisk, the total bandwidth increased linearly as the number of cores increased. When the storage device was an NVMe SSD, `ijournaling` showed a linear improvement in the total bandwidth at up to four cores. When more than four cores were used, however, the rate of bandwidth increase was reduced owing to the bandwidth limit of the SSD.

6.4 Benchmark Results

The fsync latency can affect the performance of an application if frequent fsync system calls are generated. To evaluate the performance gain from `ijournaling`, several benchmark programs were used. Figure 10 shows the results of Mobibench [16], which was designed for testing the SQLite performance on an Android-based smartphone. Because SQLite DBMS generates frequent fsync calls, its performance is closely related to the fsync latency. One-thousand DB transactions were generated, and two DB journaling modes, i.e., WAL journal and rollback journal modes, were used. The fio background application was optionally executed using BG_{bw} = 30 MB/s. We measured the performance improvement over the normal journaling scheme.

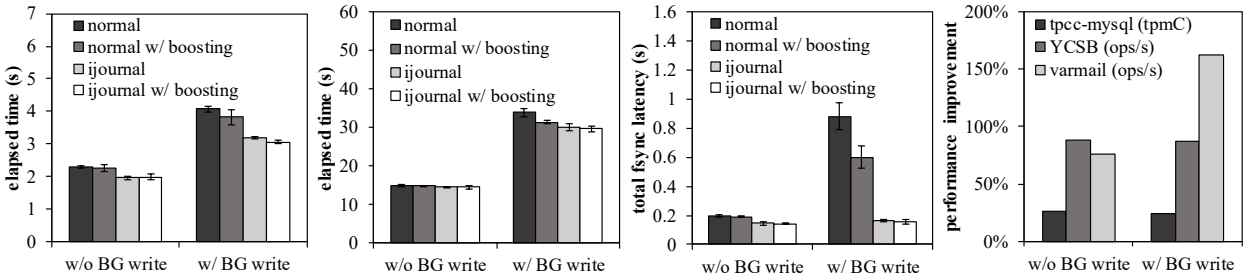
Even when no background process was used, and therefore no CTX dependency occurred, `ijournaling` improved the DB performance. The performance gain in WAL journal mode is due to the reduced journal write traffic of `ijournaling`. Whereas normal journaling must write multiple metadata blocks in a journal, `ijournaling` writes only two ijournal blocks for most cases because the modified inode entry is put into a 4 KB ijournal header block. The significant performance gain in rollback journal mode resulted from the CTX dependency problem. Although no background process was executed, the SQLite updated multiple files and the rollback journal file was truncated for every DB transaction. Owing to the truncated file, a *discard* command was included in the normal transaction. Therefore, the transaction commit was delayed owing to the handling of the *discard* command in normal journaling.

When a background process was executed, `ijournaling` showed significant performance improvements. In normal journaling, a journal commit invoked by an fsync call flushed about 25 MB of data blocks owing to the CTX dependency problem. The improvements achieved through boosting were poor because the SQLite application calls an `fsync()` immediately after a `write()` operation.

Figures 11(a) and (b) compares the performances of two smartphone applications under different journaling schemes. The camera burstshot program took 20 photos, and the application install program installed Angrybird. The fio background application was optionally executed using BG_{bw} = 30 MB/s. These applications also delete several files, and thus the transaction committed by an `fsync()` includes *discard* commands. Therefore, the `ijournaling` scheme reduced the execution times even when no background application was running. When a background application was executed, the performance improvements by `ijournaling` were more significant. Because the application install program is computing-intensive owing to the compilation work for java class files, its execution time is not significantly affected by the file-system performance. When we observed only the fsync latencies, however, there were significant performance gains by `ijournaling`, as shown in Figure 11(c).

Figure 11(d) shows the performance improvements by `ijournaling` for the desktop benchmarks. Three workloads were used: Percona's tpcc-mysql [5], YCSB [11], and FileBench's varmail [2]. In the tpcc-mysql workload, the DB page size was configured to 4 KB, ten warehouses were used, 16 connections were applied, and the running time was 100 seconds. In the case of the YCSB workload, the MySQL system and a update-heavy workload (i.e., Workload A), which has 50% reads and 50% updates, were used. The varmail workload was run with the default option. The fio background application was optionally executed using BG_{bw} = 200 MB/s.

Even when no background application was used, `ijournaling` improved the performance on the desktop benchmarks because these workloads generated multiple concurrent threads that called an `fsync()` simultane-



(a) smartphone burstshot (b) smartphone app install (c) app install (fsync latency) (d) desktop benchmark

Figure 11: Real application benchmarks.

ously, and thus the inter-transaction dependency was severe. For example, for the tpcc-mysql workload, 28.3% of all fsync calls were delayed owing to the IT dependency. In addition, `ijournaling` reduced the journal write traffic by 56% owing to its file-level journaling scheme for the YCSB workload. Most of the transactions committed by `fsync()` had *discard* commands in the varmail workload.

When a background application was executed, there were no further performance improvements compared with the case of no background process for the tpcc-mysql and YCSB workloads because these workloads had an excessive inter-transaction problem. The varmail workload is more fsync-intensive. In the case of the varmail workload, while a JBD2 was committing a normal transaction, many fsync calls were delayed owing to the IT dependency problem in normal journaling. Therefore, the performance gain by `ijournaling` was more significant.

6.5 Crash Recovery Tests

Finally, we conducted crash recovery tests under four file-system modification scenarios. During each test scenario, a crash was triggered and the system was restarted. The file-system operations generated during the tests were printed out, and recorded on a monitoring computer. The required file-system changes were derived from the logs, and we were able to check whether the file-system changes were correctly recovered. In addition, we also checked the file-system consistency using the `e2fsck` utility.

In the first scenario, one-thousand files were created sequentially, among which only odd-numbered files were fsynced. A system crash was triggered before normal periodic journaling was invoked. This scenario was able to test whether `ijournaling` can recover the inodes of fsynced files and whether the recovered directory entry of the parent directory has the entries of only committed files. In the second scenario, a file was created, and 4 KB of data were appended to the file repeatedly. After each 4 KB write, an fsync was called. To make external extent blocks, a crash was triggered after the file size reached larger than 1 GB. This test covered the correctness of external extent tracking. For the third scenario, more than two depths of directories were made, and an fsync for a file at leaf node was called. This scenario was able to check whether all related parent directories were recovered. For the last scenario, ten threads were generated, each of which executed file operations randomly selected among `mkdir`, `create`, `write`, `truncate`, `unlink`, and `fsync`. For each of these scenarios, we ascertained that `ijournaling` can correctly recover the fsynced files and their related directories without any file-system inconsistencies.

7 Conclusion

We rely on the journaling of data updates for file-system consistency, and synchronous writes for data durability. However, latency-sensitive synchronous operations such as an `fsync()` system call can be delayed under the compound transaction scheme of the current journaling technique. Because a compound transaction includes irrelevant data and metadata, as well as those of fsynced file, the fsync latency can be unexpectedly long. In this paper, we first analyzed the affecting factors that may delay an fsync operation, and proposed a novel hybrid journaling technique, called `ijournaling`, which journals only the related file-level transactions of an fsync call and recovers the file-system consistency through file-level journals upon a crash recovery. Experiments using real devices showed that there are significant improvements to the fsync latencies when using `ijournaling`, and that many synchronous applications can benefit from the proposed `ijournaling` technique.

Acknowledgements

We would like to thank Theodore Ts'o, who was our shepherd, and anonymous reviewers for their valuable comments and suggestions. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP). (No. 2016R1A2B2008672)

References

[1] Ext4 filesystem. https://www.kernel.org/doc/Documentation/filesystems/ext4.txt.

[2] Filebench. http://filebench.sourceforge.net/.

[3] SQLite. https://sqlite.org.

[4] Sysbench. https://github.com/akopytov/sysbench.

[5] tpcc-mysql. https://github.com/Percona-Lab/tpcc-mysql.

[6] J. Axboe. FIO - flexible IO tester. http://freshmeat.net/projects/fio/.

[7] Jeff Bonwick and Bill Moore. ZFS: The last word in file systems. http://wiki.illumos.org/download/attachments/1146951/zfs_last.pdf, 2007.

[8] Li-Pin Chang, Po-Han Sung, and Po-Hung Chen. Fast file synching for applications in flash-based android devices. In *Proceedings of the 3rd Non-Volatile Memory Systems and Applications Symposium*, pages 1–6, 2014.

[9] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP'13, pages 228–243, 2013.

[10] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency without ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 101–116, 2012.

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC'10, pages 143–154, 2010.

[12] Jonathan Corbet. Solving the ext3 latency problem. http://lwn.net/Articles/328363/.

[13] Rasha Eqbal. ScaleFS: A multicore-scalable file system. Master's thesis, Massachusetts Institute of Technology, August 2014.

[14] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1993.

[15] Daeho Jeong, Youngjae Lee, and Jin-Soo Kim. Boosting quasi-asynchronous I/O for better responsiveness in mobile devices. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 191–202, 2015.

[16] Sooman Jeong, Kisung Lee, Jungwoo Hwang, Seongjin Lee, and Youjip Won. AndroStep: Android storage performance analysis tool. In *Software Engineering Workshops*, pages 327–340, 2013.

[17] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O stack optimization for smartphones. In *Proceedings of the 2013 USENIX Annual Technical Conference*, ATC'13, pages 309–320, 2013.

[18] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. SpanFS: A scalable file system on fast storage devices. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, ATC'15, pages 249–261, 2015.

[19] Yunji Kang and Dongkun Shin. Per-block-group journaling for improving fsync response time. In *Proceedings of the 18th IEEE International Symposium on Consumer Electronics*, pages 22–25, 2014.

[20] Jan Kara. ext4: fix data exposure after a crash. https://patchwork.kernel.org/patch/9156691/, 2016.

[21] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Physical disentanglement in a container-based file system. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'14, pages 81–96, 2014.

[22] Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Marshall Kirk Mckusick. Ffsck: The fast file-system checker. *ACM Transactions on Storage*, 10(1):2:1–2:28, 2014.

[23] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: Current status and future plans. In *Proceedings of the Ottowa Linux Symposium*, 2007.

[24] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding manycore scalability of file systems. In *Proceedings of the 2016 USENIX Annual Technical Conference*, ATC'16, pages 71–85, 2016.

[25] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. *ACM Transactions on Computer Systems*, 26(3):6:1–6:26, 2008.

[26] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *Proceedings of the 2005 USENIX Annual Technical Conference*, ATC'05, pages 105–120, 2005.

[27] Kai Shen, Stan Park, and Meng 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.

[28] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the xfs file system. In *Proceedings of the 1996 USENIX Annual Technical Conference*, ATC'96, pages 1–14, 1996.

[29] Theodore Ts'o. ext4: remove calls to ext4_jbd2_file_inode() from delalloc write path. http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=f3b59291a69d0b734be1fc8be489fef2dd846d3d, 2012.

Scaling Distributed File Systems in Resource-Harvesting Datacenters*

Pulkit A. Misra[⋆] *Íñigo Goiri*[†] *Jason Kace*[†] *Ricardo Bianchini*[†]
[⋆]*Duke University* [†]*Microsoft Research*

Abstract

Datacenters can use distributed file systems to store data for batch processing on the same servers that run latency-critical services. Taking advantage of this storage capacity involves minimizing interference with the co-located services, while implementing user-friendly, efficient, and scalable file system access. Unfortunately, current systems fail one or more of these requirements, and must be manually partitioned across independent subclusters. Thus, in this paper, we introduce techniques for automatically and transparently scaling such file systems to entire resource-harvesting datacenters. We create a layer of software in front of the existing metadata managers, assign servers to subclusters to minimize interference and data movement, and smartly migrate data across subclusters in the background. We implement our techniques in HDFS, and evaluate them using simulation of 10 production datacenters and a real 4k-server deployment. Our results show that our techniques produce high file access performance, and high data durability and availability, while migrating a limited amount of data. We recently deployed our system onto 30k servers in Bing's datacenters, and discuss lessons from this deployment.

1 Introduction

Each datacenter costs billions of dollars to build, populate, and operate. Even though procuring servers dominates this cost [5], servers are often poorly utilized, especially in clusters that host interactive services [5, 9].

Resource harvesting. The co-location of useful batch workloads (*e.g.*, data analytics, machine learning) and the data they require on the same servers that run interactive services is effective at extracting more value from the servers, and at reducing the overall number of servers that must be procured. Effectively, the batch workloads can harvest the spare cycles and storage space left by the services. However, the services must be shielded from any non-trivial performance interference produced by the batch workloads or their storage accesses; prior work [18, 19, 26] has addressed this problem. At the same time, we must ensure that the services' resource requirements and management do not unnecessarily degrade batch workloads' performance or compromise the availability and durability of the batch workloads' data.

*Pulkit Misra was a summer intern at Microsoft Research.

Zhang *et al.* [26] built a resource-harvesting distributed file system for the batch workloads' data that achieves these characteristics by smartly placing the file block replicas across the servers. However, they did not address how to scale the file system for full datacenter-wide (*e.g.*, 50k servers) harvesting, which is our target.

Scaling distributed file systems. Most distributed file systems have not been designed to scale (transparently or at all) to such large sizes. For example, HDFS [2], Cosmos Store [7], and GFS [12] do not typically scale well beyond a few thousand servers, as they rely on a centralized metadata manager (with *standby* replicas). To scale beyond this size, administrators must create separate subclusters (of whatever maximum size can be efficiently handled), each running an independent manager for an independent portion of the namespace.

Unfortunately, this approach to scaling has several drawbacks. First, users are presented a partitioned view of the namespace and often have full control over which subcluster to place their folder/files in. Second, exercising this control, users may inadvertently fill up a subcluster or overload it with a high access demand. Third, to mitigate these situations, administrators must manage folder/file placement manually (via folder/file migration and/or forcing users to the more lightly used subclusters). Fourth, it is very difficult for administrators (and impossible for users) to understand the characteristics of the co-located services in each subcluster well enough to make appropriate folder/file placement decisions, especially as services are complex and numerous.

Another approach to scaling is to implement multiple, strongly consistent, *active* metadata managers, *e.g.* [1, 23]. Though no information has been published about Google's Colossus (the follow-on to GFS), we understand that it implements such managers. However, this approach also has two key drawbacks: (1) the system becomes more complex, and this complexity is only required for large installations (simpler systems that also work well for more popular smaller systems are preferrable); and (2) any software bugs, failures, or operator mistakes have a greater impact without the isolation provided by subclusters, as highlighted in [22].

Given the drawbacks of these two approaches, it is clear that a cleanly layered, automated, and manageable approach to distributed file system scalability is needed.

Our work. Thus, in this paper, we design techniques for

automatically and transparently scaling file systems to entire resource-harvesting datacenters with tens of thousands of servers. We use Zhang's replica placement algorithm within each subcluster, but focus on how to "federate" the subclusters transparently and efficiently. We achieve these high-level characteristics by inserting a layer of software between clients and metadata managers that understands the federated namespace and routes requests to the appropriate subclusters.

Moreover, our techniques seek to (1) avoid interference from co-located services; and (2) promote behavioral diversity, good performance, and good space usage across subclusters. Achieving these goals at the same time is challenging, given the large number of services and servers, and the widely varying folder/file size and access characteristics of the batch workloads.

To simplify the problem, we divide it into two parts. First, we select the servers to assign to each subcluster in a way that maximizes data availability and durability. Specifically, we use consistent hashing [17] for this new purpose, which has the added benefit of limiting data movement when resizing subclusters. Second, we assign folders/files to subclusters and efficiently migrate them when either a subcluster starts to run out of space, or a subcluster's metadata manager starts to receive an excessive amount of access load. We model this rebalancing as a Mixed Integer Linear Program (MILP) problem that is simple enough to solve efficiently. Migrations occur in the background and transparently to users.

Implementation and results. To explore our techniques concretely, we build them into HDFS and call the resulting system "Datacenter-Harvesting HDFS" or simply "DH-HDFS". We selected HDFS because (1) it is a popular open-source system that is used in large Internet companies, *e.g.* Microsoft, Twitter, and Yahoo (we are contributing our system to open source [16]); (2) our target workloads are mostly analytics jobs over large amounts of data, for which HDFS provides adequate features and performs well; and (3) many data-analytics frameworks, like Spark and Hive, can run on HDFS.

Our evaluation uses a real deployment in a production datacenter, real service and file access traces, and simulation of 10 real datacenters. The results show that our server-to-subcluster assignment prevents interference from co-located services, minimizes data movement when subclusters are added/removed, and promotes data availability and durability for batch jobs. The results also show that our folder migration policy is efficient, migrating a small percentage of the folders; just enough to manage severe space shortages or load imbalances. When combining our techniques, DH-HDFS improves durability and availability by up to 4 and 5 orders of magnitude, respectively, compared to prior approaches. Finally, the results show that the federation layer imposes little performance overhead.

Production use. We currently have 4 DH-HDFS deployments in production use in our datacenters; the largest deployment now has 19k+ servers spread across 6 subclusters. We discuss lessons from these deployments.

Implications for other datacenter types. Though we focus on resource-harvesting datacenters, some aspects of our work also apply to scenarios where the batch workloads have the same priority over the resources as the co-located services, or where there are no co-located services. Specifically, our federation architecture and techniques for folder/file mapping to subclusters with periodic migrations apply to any scenario. Our technique for server-to-subcluster mapping would work in other scenarios, but is not strictly needed.

Summary. Our main contributions are:

- We propose novel techniques for scaling distributed file systems while accounting for data durability, availability, storage capacity, and access performance in large resource-harvesting datacenters. In particular, we introduce (a) layering for transparent scalability of unmodified existing systems; (b) consistent hashing for subcluster creation; and (c) MILP-based dynamic file migration.
- We implement our techniques in HDFS to create DH-HDFS, which we have deployed in production.
- We evaluate our techniques and system, using real workloads, real experimentation, and simulation.
- We discuss lessons from DH-HDFS in production use.

2 Background and related work

2.1 Resource-harvesting datacenters

In resource-harvesting datacenters, most servers are allotted to native, often latency-critical, workloads. Because of their latency requirements, these workloads store data using their servers' local file system. We refer to these workloads as *"primary tenants"*. To improve utilization, lower priority workloads called *"secondary tenants"*, such as batch data analytics jobs, can harvest any spare capacity left idle by primary tenants. Primary tenants have priority over their servers' resources, *i.e.* a load spike may cause secondary tenants to be throttled (or even killed) and their storage accesses to be denied. Moreover, primary tenant developers own their servers' management, *i.e.* they are free to perform actions that destroy disk data. Among other scenarios, disk reimaging (reformatting) occurs when developers re-deploy their primary tenants from scratch, and when the management system tests the resilience of production services.

The resource-harvesting organization is reminiscent of large enterprises where different departments have their own budgets, without a central infrastructure group. Nevertheless, multiple Internet companies, such as Microsoft and Facebook, use this type of underlying system.

Though Google's infrastructure is fully shared, *i.e.* any workload is treated the same, large Google tenants may also request priority over their allotted resources [24].

2.2 Diversity-aware replica placement

A challenge in harvesting is protecting file block availability and durability: (1) if we store all of a block's replicas in primary tenants that load-spike at the same time, the block may become unavailable; (2) if developers or the management system reimage the disks containing all of a block's replicas in a short time span, the block may be lost. Thus, a replica placement algorithm must account for primary tenant and management activity.

Zhang's placement algorithm [26] places replicas within a single cluster (*i.e.*, a few thousand servers), while maximizing diversity: it does not allow multiple replicas of a block to be placed in any logical (*e.g.*, servers of the same primary tenant) or physical (*e.g.*, rack) server grouping that induces correlations in resource usage, disk reimaging, or failures.

We build upon Zhang's single-cluster work by creating a federation of clusters (we refer to each cluster in the federation as a subcluster). In this context, we also select which servers to assign to each subcluster, and automatically rebalance space and access load across subclusters.

2.3 Large-scale distributed data storage

Large-scale file systems. Several distributed file systems (*e.g.*, [1, 20, 23]) have been proposed for large installations. Though potentially scalable, they involve complexity and overhead in metadata management, and are hard to manage and maintain in large-scale production. Moreover, they are often optimized for general workloads, and not those of datacenter applications (*e.g.*, write-once, append-only).

For these reasons, file systems at Microsoft [2, 7], Facebook [2, 6], Twitter [2], and other datacenter operators are much simpler. They rely on a centralized metadata manager (*e.g.*, "Namenode" in HDFS) that hosts all metadata, handles all metadata accesses, and tracks the storage nodes. To scale, administrators manually partition the overall file set into independent file systems, each in a subcluster. Some systems (*e.g.*, ViewFS [3], Cosmos Store [7]) enable users to access multiple subclusters transparently, by exploiting "mount tables" that translate folder names to subclusters. However, the client-local mount tables are independent and not kept coherent. In contrast, Google's Colossus is rumored to implement multiple active metadata managers. This approach is more complex and does not benefit from the fault- and mistake-isolation provided by subclusters [22].

Rebalancing. Some systems rebalance metadata across metadata managers without subclusters, *e.g.* [1, 23].

A challenge with subclusters is that they may become imbalanced in terms of space usage and/or access load. In resource-harvesting datacenters, imbalance is also possible in the primary tenants' resource usage and management behavior; *e.g.*, changes in primary tenant disk reimaging patterns could start to harm a subcluster's durability. We are not aware of prior policies for folder/file rebalancing across subclusters.

Several works considered rebalancing within a single cluster. For example, [13] and [14] proposed balancing the access load. Considering space and access load, Singh *et al.* [21] accounted for multiple resources (*e.g.*, switches, disks) in making greedy rebalancing decisions. The built-in HDFS rebalancer can be used to manually rebalance data, and to populate newly added servers.

HDFS Federation. HDFS has an option to split the namespace (and block management) explicitly across independent metadata managers, while storing data in any server [11]. This approach does not involve subclusters, but exposes multiple namespaces that users must manage manually [3], and limits scaling as all servers still heartbeat to all managers. *Our system is quite different,* and should not be confused with this HDFS option.

3 Federation architecture

3.1 Overview

Our architecture assumes an *unmodified* underlying distributed file system similar in structure to HDFS [2], Cosmos Store [7], and GFS [12]. It federates subclusters of the distributed file system, each defined by its own metadata manager, data storage nodes, and client library.

Each subcluster operates independently, unaware of other subclusters. This characteristic simplifies our design, and means that all replicas of a file block live in the same subcluster. As we illustrate in Figure 1, we interpose a highly available and fault-tolerant layer of software between the clients and the subclusters' metadata managers (labeled "MM" in the figure). The layer comprises (1) multiple client request routers (labeled "R"); (2) a state store that maintains a *global mount table* (*i.e.*, the folder/file-to-subcluster mappings, which we call "mount table entries" or simply "mount points") and other pieces of state about the federation; and (3) a folder/file rebalancer. Next, we detail these components.

3.2 Client request routers

The routers transparently expose a single global namespace to the clients through the standard metadata manager interface of the underlying distributed file system. Clients are unaware of the routers. A client's file access may reach any router (arrow #1 in Figure 1), as routers may sit behind a load balancer or some other request distribution mechanism. The router then consults the state store to determine the metadata manager for the proper

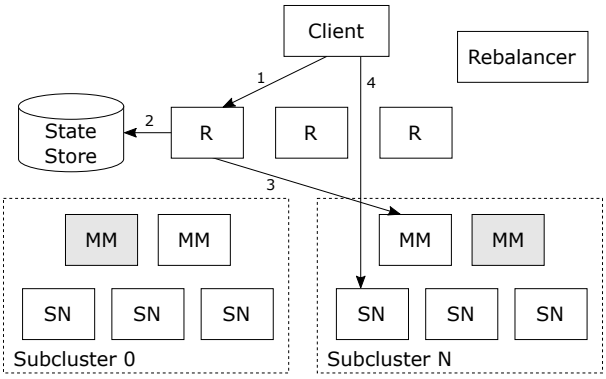


Figure 1: Federation layer comprising transparent request routers, a logically centralized state store, and a folder/file rebalancer. R = router; MM = metadata manager; SN = storage node; grey color = standby manager.

subcluster (arrow #2), and forwards the request to it (arrow #3). The reply from the manager flows back in the opposite direction. The reply lists the address of the storage nodes for all replicas of the file's blocks, so the client can communicate directly with the corresponding storage nodes on actual block accesses (arrow #4).

The routers intercept all calls. Most calls involve simply forwarding the same parameters to the proper metadata manager, perhaps after adjusting any pathnames. However, four types of calls may require additional processing: renames, deletes, folder listings, and writes. Routers fail any renames or deletes of mount points, like in other file systems (*e.g.*, Linux). Renames or deletes of folders/files that only affect one subcluster can simply be forwarded to the proper metadata manager. We handle renames of folders/files that affect multiple subclusters by performing the rename in the state store (*i.e.*, creating a new mount table entry) and having the rebalancer migrate the data later. Importantly, the routers "lock" the federated namespace during these renames to prevent inadvertent cycles [10]. Folder listing involves contacting the parent folder's subcluster, and including any mount points under the same parent folder. Finally, routers may fail folder/file writes during short periods, to guarantee consistency (*e.g.*, during rebalancing operations, as we discuss in Section 3.4).

To avoid frequent communication with the state store, the routers *cache* the folder/file-to-subcluster mappings locally. The router's cache entries may become stale, as a result of rebalancing or of losing contact with the state store for a period of time. To prevent uses of stale entries, we ensure all routers acknowledge mount table changes, and check the state store for freshness of their entries.

Dependability. Other than the disposable cache state, routers are stateless and can be replaced or restarted for high availability and fault tolerance. The routers send heartbeats to the state store, including information about metadata managers and subclusters. If a router cannot heartbeat for a period T, it enters a "safe" mode (no accesses allowed), and the other routers take on the full metadata access load. If a router does not update its status for a period $2T$, any locks it holds are taken away.

The router uses standard HDFS interfaces to query the availability of the redundant managers and the space available in the subcluster. For dependability, we associate routers with overlapping sets of metadata managers, and resolve conflicts using quorum techniques.

3.3 State store

The store maintains four pieces of state about the federation: (1) the global mount table; (2) the state of the routers; (3) the access load, available space, and availability state of the metadata managers/subclusters; and (4) the state of rebalancing operations (Section 3.4).

The mount table contains explicit mappings of folders and files to subclusters. For example, there could be a mapping from folder `/tmp/` to subcluster 3 in folder `/3/tmp/` in the federated namespace. Only the system administrators or the rebalancer can create or modify entries in the mount table. But, since there may be multiple concurrent accesses to it, writes to the mount table must be properly synchronized. In terms of structure, the logically centralized nature of the state store simplifies our architecture. However, for larger installations (*e.g.*, tens of active routers), the store must be physically distributed and provide strong consistency. Existing systems, *e.g.* Zookeeper [15], provide these features and can be used to implement the store. We use Zookeeper for our implementation (Section 5).

3.4 Rebalancer

Subclusters may be unbalanced in three ways: (1) the characteristics of their primary tenants are such that some subclusters do not exhibit enough diversity for high-quality replica placement; (2) the access load they receive may be skewed and overload some metadata managers or interfere with the primary tenants in some subclusters; and/or (3) the amount of data they store may be widely different, threatening to fill up some of them.

We address the first way with our server-to-subcluster mapping (Section 4.1). To address the other ways, our architecture includes a rebalancer component. The rebalancer migrates folders/files across subclusters (folders/files and subclusters are selected as discussed in Section 4.2) and then updates the mount table. The source data may be a sub-path of an existing mount point, *i.e.* the rebalancer can create new mount table entries.

Ensuring consistency. The rebalancer must ensure the consistency of the federated file system, as regular client traffic may be directed to the files it is migrating, and multiple failure types may occur. To achieve this, it first

records in the state store a write-ahead log of the operations it is about to start. As each operation completes, it updates the log to reflect the completion. A failed rebalance can be finished or rolled back using the log. The log also protects the system against inadvertently running multiple concurrent rebalancer instances: each instance checks the log before a migration, and aborts if it finds that another instance is actively altering the same part of the namespace.

Second, it takes a write lease on the corresponding mount table entries (it may need to renew the lease during long migrations) and records the state (*e.g.*, last modification time) of the entire subtree to be migrated. The lease prevents changes to the mount table points (by administrators or multiple instances of the rebalancer), but not to the source folders/files themselves by other clients.

Third, the rebalancer copies the data to the target subcluster. At the end of the copy, it checks whether the source data was modified during the copy. Via the state store, the rebalancer must instruct the routers to prevent writes to the source and target subtrees (and wait for routers to acknowledge), before it can compare the metadata for the subtrees. This prevents a client from modifying the source data after it has been inspected for recent changes, but before the mount table has been updated to point to the target subcluster. During this checking phase, clients are still able to read from the source data. If the source data is unchanged, the rebalancer updates the mount table, waits for all routers to acknowledge the change (at which point the source data can no longer be accessed), stops blocking writes to the source and target subtrees, and then removes the data from the source subcluster. If the source data was modified during the copy, the rebalancer rolls back and either re-starts the entire copy or simply re-copies the changed files. Our current implementation takes the latter approach. Similarly, a failure in any step of the rebalancer (*e.g.*, a file migration) causes a roll back. The rebalancer tries to complete the copy a few times (three times in our implementation). If these re-tries are not enough to complete the copy, the rebalancer rolls back but, this time, it blocks writes to the data before the copy starts.

Fourth, when the migration successfully completes, the rebalancer gives up the lease on the mount points.

3.5 Alternative architecture we discarded

We considered simply extending a system like ViewFS [3] with a shared mount table, but this design would not be transparent to the underlying file system; it would require changing the file system's client code to implement the functionality of our routers.

4 Federation techniques

In this section, we discuss the two tiers of techniques we propose to simplify the problem of organizing the federated file system in a resource-harvesting datacenter: (1) server-to-subcluster mapping, (2) folder/file-to-subcluster mapping and dynamic rebalancing. The first tier statistically guarantees that subclusters are diverse in terms of primary tenants' resource usage and disk reimaging behaviors. The second tier ensures that no subcluster undergoes an excessive access load or a storage space shortage due to secondary tenants, while other subclusters have available capacity. We finish the section with a discussion of alternative techniques.

4.1 Assign servers to subclusters

The components above provide the mechanisms we need to create and manage the federated file system. However, we still need a policy for assigning servers to subclusters in the first place. We have multiple goals for this policy:

1. Ensure that subcluster addition/removal (*e.g.*, when the administrator adds the servers of a large primary tenant into the harvesting infrastructure) does not cause massive data reorganization;
2. Promote network locality within subclusters;
3. Produce diversity in primary tenants' resource usage and reimaging behaviors in each subcluster for high availability and durability; and
4. Produce subclusters with balanced primary tenant storage space usage. (The rebalancer balances the subclusters with respect to secondary tenant access load and space consumption.)

To achieve these goals, we first define the number of subclusters as the total number of servers divided by the number of servers that can be efficiently accommodated by a metadata manager ($\sim$4000 servers per subcluster by default). Then, our policy leverages consistent hashing [17] of rack names for assigning server racks to subclusters. As consistent hashing is probabilistic, each subcluster is assigned multiple virtual nodes [8] on the hash ring to balance the number of servers assigned to each subcluster. Consistent hashing reduces the amount of data reorganization needed when subclusters are added to/removed – goal #1 above. We hash full racks to retain within-rack network locality (within a datacenter, there may be hundreds of racks, each with a few dozen servers and a top-of-rack switch) – goal #2. Finally, since each primary tenant is spread across racks for fault tolerance and most primary tenants are relatively small, randomizing the rack assignment to subclusters statistically produces evenly balanced diversity in primary tenant load and reimaging behaviors, as well as balanced space usage – goals #3 and #4. We are unaware of other work that has used consistent hashing for this purpose.

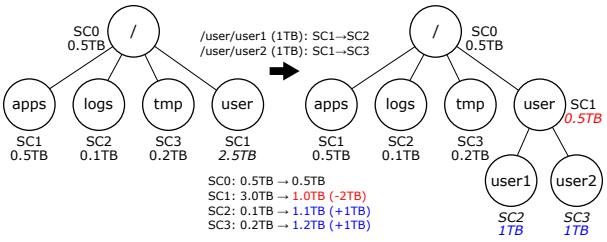


Figure 2: Example of rebalancing due to storage capacity. The file system is stored across 4 subclusters. After a rebalance, `/user` is split across 2 subclusters.

4.2 Assign/rebalance files to subclusters

Using the technique above, we now have subclusters with statistically balanced primary tenant resource usage, disk reimaging, and space usage. But we still need to assign the secondary tenants' folders/files to them, and possibly re-assign (rebalance) folders/files, when some of the current assignments are no longer appropriate.

Creation-time assignment policy. To avoid creating new mount points every time a new folder/file is created, the routers forward create operations to the same subcluster of the parent folder. This may eventually fill up some subclusters while others have plenty of free space. In addition, it may produce subclusters that receive high access loads while others receive much lower loads. We leverage rebalancing to correct these situations.

Rebalancing policy. The rebalancer wakes up periodically (*e.g.*, hourly) and compares the recent subclusters' metadata access load and free space to pre-defined watermark thresholds. We opt not to rebalance (an expensive operation) simply because the subclusters are imbalanced with respect to load or space. Instead, the thresholds define values beyond which a subcluster would be considered "under stress". Each rebalancing round tries to bring all subclusters below the watermarks. Administrators can also start a rebalancing round manually.

The rebalancer finds the subclusters' information in the state store. As the routers intercept all accesses to the metadata managers, they can easily accumulate this information and store it in the state store. (The routers cannot determine the access load imposed on a subcluster's storage nodes, only that on its metadata manager. Nevertheless, the manager is the first to overload, since it is centralized.) The routers periodically consult the managers to find out the amount of free space in each subcluster, and store it in the state store during heartbeats.

Figure 2 illustrates an example where subcluster 1 is highly loaded. The rebalancer decides to split `/user` and spread it across subclusters 2 and 3.

Rebalancing as optimization. To determine which folders/files to migrate to which subclusters, we model rebalancing as a MILP problem and use a standard solver for it. MILP is expressive enough and works well for our constraints and objectives. We are not aware of similar approaches to file system rebalancing.

We start by creating a representation of the federated namespace, where we annotate each tree node with (1) the peak amount of load it has received in any short time interval (*e.g.*, 5 minutes) since the last rebalance, (2) the current size of the subtree below it, and (3) its current subcluster. We prune nodes that exhibit lower load and lower size than corresponding administrator-defined low-end thresholds. This limits the size of the MILP problem, making it efficient to solve.

We use the pruned tree as the input to the MILP problem. The main constraints are the maximum access load a metadata manager can handle, and the maximum storage capacity of each subcluster. As its outputs, the solution produces the subcluster in which each of the tree nodes should be after rebalancing. As the objective function, we minimize a utility function combining several weighted factors: access load per subcluster, used storage capacity per subcluster, amount of data to move in the rebalance, and the number of entries in the mount table after rebalancing. The administrator is responsible for defining the weight for each factor.

Since these factors are measured in different units, we represent them as percentages over their corresponding watermarks. We introduced the access load and used capacity thresholds above. For the amount of data to move, we compute the percentage with respect to the combined size of the files that need to move to bring all subclusters below the watermarks. For the number of mount table entries, we use the percentage compared to the maximum between the number of subclusters and the number of folders in the first level of the federated namespace.

Besides its ability to derive efficient rebalances, our optimization approach is flexible in that different objective functions and constraints can be easily implemented.

4.3 Alternative techniques we discarded

Assigning servers and files to subclusters at once. For the file system organization, we considered solving the entire server and folder/file assignment problem as a large mathematical program, including primary tenants' characteristics and the federated file system. Doing so would be unwieldy; splitting the problem into two tiers of techniques makes the problem manageable.

Assigning servers to subclusters. We considered random assignment per server, per primary tenant, and per groups of primary tenants. These approaches produce subclusters with high diversity, but cause significant data movement when a subcluster is added/removed. Consistent hashing achieves diversity without this problem.

Assigning/rebalancing files to subclusters. We considered using consistent hashing of file names. There are two main problems with this approach: (1) the files in

each folder could be spread across multiple subclusters, leading to a very large mount table; and (2) a subtree rename would likely cause the entire subtree to move. Using consistent hashing of immutable file identifiers [10] would solve the latter problem but not the former.

5 Implementation and deployment

We implement our federation architecture and techniques in HDFS, and call the resulting system "Datacenter-Harvesting HDFS" or simply "DH-HDFS". We are contributing our system to open source [16].

In terms of structure and behavior, HDFS matches the underlying distributed file system in Figure 1: its metadata manager is called "Name Node" (NN) and its per-server block storage node is called "Data Node" (DN). The NN implements all the APIs of standard distributed file systems, and maintains the namespace and the mapping of files to their blocks. The (primary) NN is backed up by one or more secondary NNs. In our setup, the NN replicates each block (256 MBytes) three times by default. On a file access, the NN informs the client about the servers that store the replicas of the file's blocks. The client contacts the DN on one of these servers directly to complete the access. The DNs heartbeat to the NN; after a few missing heartbeats from a DN, the NN starts to recreate the corresponding replicas in other servers without overloading the network (30 blocks/hour/server). Within each subcluster, we use Zhang's replica placement algorithm [26] to achieve high data durability and availability in a resource-harvesting datacenter.

We place the routers behind a load balancer and configure clients (via their standard configuration files) to use the load balancer address as the NN. We implement the state store using Zookeeper [15]. At a high level, our router and state store organization purposely matches a similar architecture for YARN federation [4]. The rebalancer runs as a separate MapReduce program (one file per map task). For scalability, each DN determines its subcluster membership independently at startup time. If it needs to move to a different subcluster, the DN first decommissions itself from the old subcluster and then joins the new one. We also allow administrators to define the membership and trigger rebalances manually.

Based on our experience with the system, we define the number of subclusters as the number of servers in the datacenter divided by 4k (the largest size that HDFS handles efficiently in our setup). We set the routers to heartbeat to the state store every 10 seconds by default. In addition, we define the threshold for access load as an average 40k requests/second (near the highest throughput that an NN can handle efficiently in our setup) over any 5-minute period, and the space threshold as 80% of each subcluster's full capacity. We leverage an HDFS utility (DistCP) for copying file system subtrees. If writes occur during a copy, DistCP only re-copies the individual files written. It also transparently handles failures of NNs and DNs. We configure the rebalancer to try a subtree copy 3 times before re-trying with blocked client writes (Section 3). All settings above are configurable.

6 Evaluation

6.1 Methodology

Workloads. To represent the primary tenants, we use detailed CPU utilization and disk reimaging statistics of all the primary tenants (thousands of servers) in 10 real large-scale datacenters.[1] As our secondary tenants' file access workload, we use a real HDFS trace from Yahoo! [25]. The trace contains 700k files and 4M file accesses with their timestamps. The trace does not specify file sizes, so we assume each file has 6 blocks, for a total of 4.2M blocks (as we replicate each block 3 times, the total dataset is 3PB). The trace does not specify whether file access operations are for reading or writing, so we assume that each create operation represents a full file write and each open operation represents a full file read. This assumption is accurate for systems like HDFS, which implement write-once, read-many-times files. Overall, our trace contains 3.7M reads and 0.3M writes.

Simulator. Because we cannot experiment with entire datacenters and need to capture long-term behaviors (*e.g.*, months), we extend the simulation infrastructure from [26] to support multiple subclusters. We faithfully simulate the CPU utilization and reimaging behavior of the primary tenants, federation architecture, the techniques for server and folder/file assignment, and HDFS with diversity-aware replica placement. In the simulator, we use the same code that implements server assignment to subclusters, data placement and rebalancing in our real systems. For simplicity, we simulate each rebalance operation as if it were instantaneous (our real system experiments explore the actual timing of rebalances). The simulator replays the logs from the 10 datacenters for simulating the primary tenants' CPU utilization and disk reimages, and uses the Yahoo! trace [25] for simulating the secondary tenants' block accesses. All operations are faithfully executed based on their logged timestamps.

The simulator outputs durability (percentage of blocks retained, despite disk reimages), availability (percentage of successful accesses, despite primary tenant resource usage), usable space, access load for each subcluster, and amount of data migrated. For our durability results, we simulate 6 months of the primary tenants' reimages. For our availability results, we simulate 1 month of primary tenants' utilizations and repeat the Yahoo! trace over this

[1] Due to commercial reasons, we omit certain information, such as absolute numbers of servers and actual utilizations.

period. We run each simulation 5 times and report average results. The results are consistent across runs.

For comparison with DH-HDFS, we use a baseline system that assigns servers to subclusters randomly (per group of primary tenants), which provides high diversity per subcluster. This approach represents the manual assignment we have observed in production in the absence of DH-HDFS. The baseline assigns folder/files to subclusters in such a way that each subcluster gets three levels of the federated namespace in round-robin fashion. The baseline rebalances folders/files based on a greedy algorithm (which we adapted from [21] for the federated scenario), whenever the access load or usable space exceeds their watermark thresholds. The algorithm ranks the folders and subclusters from most to least busy (in terms of load or storage), and migrates folders from the top of the folder list to subclusters from the bottom of the subcluster list. Finally, the baseline leverages Zhang's algorithm for replica placement within each subcluster.

We also present an extensive sensitivity study, exploring the impact of the load threshold, space threshold, and rebalancing frequency in DH-HDFS.

Real experiments. We use the implementation of DH-HDFS from Section 5. We run the system on 4k servers across 4 subclusters in a production datacenter. The servers have 12-32 cores and 32-128GB of memory. Each subcluster has 4 NNs and we co-locate a router on each machine that runs a NN. We use 5 Zookeeper servers for the state store. We set the rebalancer to wake up every hour. We built a distributed trace replayer to reproduce the same load as in the Yahoo! trace.

6.2 Simulation results

We start our evaluation by isolating the impact of each feature in DH-HDFS. To conserve space, these comparisons use a single datacenter (DC-7); the other datacenters exhibit similar trends. Then, we study data durability and availability of the baseline and DH-HDFS systems across the 10 datacenters. Finally, we use a sensitivity study to quantify the impact of the load threshold, the space threshold, and the rebalancing frequency.

Within-cluster replica placement. Comparing the first two rows of Table 1 isolates the impact of the replica placement approach within each subcluster. All system characteristics other than the replica placement approach are set to the baseline system, except that we turn off rebalancing. Zhang's algorithm accounts for primary tenant behaviors, whereas stock HDFS places replicas in different racks irrespective of primary tenants. The results show that Zhang's algorithm is also effective in the federated scenario: both durability and availability improve by 4 orders of magnitude. Moreover, note that losing even a single block (*i.e.*, its 3 replicas) brings durability to six 9s ($< 100 \times 1/4.2M$) in our setup, so achieving higher durability is challenging especially as primary tenants' are free to reimage collections of disks at will.

Study	Version	Dur.	Avail.
Within-subcluster replica placement	Stock HDFS	two 9s	one 9
	Diversity-aware [26]	six 9s	five 9s
Server-to-subcluster assignment	Random per primary group	two 9s	two 9s
	Random per server	six 9s	five 9s
	Random per rack	six 9s	five 9s
	Consistent hashing per rack	six 9s	five 9s
Folder-to-subcluster assignment	Round-robin per subtree (RR)	six 9s	five 9s
	RR + rebalancing from [21]	two 9s	zero 9s
	RR + our rebalancing	six 9s	five 9s

Table 1: Simulation results for DC-7.

Server-to-subcluster assignment. The next set of rows compare approaches for assigning servers to subclusters. Again, features other than server-to-subcluster assignment are those of the baseline system without rebalancing. The results show that random per primary tenant group, which groups together all primary tenants that have related functionality, performs poorly. Due to their close relationship and potentially large size, these groups do not produce enough diversity even under Zhang's algorithm. The other three approaches achieve good results, as they leverage finer grain randomization and thus benefit from primary tenant diversity.

Consistent hashing has the additional advantage of requiring limited data movement as a result of subcluster additions/removals. For example, if we were to add a new subcluster to DC-7, only 5.5% of the data would move to populate it. In contrast, the random per rack approach would move 44% of the data. On the other hand, if we were to remove the subcluster with the most data, consistent hashing would require 20% of the data to move, while random per rack would move 68% of it.

Folder-to-subcluster assignment. The following set of rows compare techniques for assigning folders to subclusters; all other features are those of the baseline system. The results show that our rebalancing approach improves performance (not shown in the table) at the same time as retaining high durability and availability. Specifically, without rebalancing, some subclusters are exposed to extremely high load; at peak, 142k accesses/second over a period of 5 minutes. With our approach, the peak load on any subcluster goes down to 38k accesses/second after rebalancing, just under our watermark threshold of 40k accesses/second. To achieve this, our rebalancer migrates 24TB of data. In contrast, the greedy rebalancer achieves a peak of 37k accesses/second, but migrates 84TB of data. Worse, this rebalancer degrades durability and availability significantly, as it does not consider the diversity of primary tenants in the lightly loaded subclusters. Had we assumed consistent hashing (instead of random per primary tenant group) for the server assignment in this comparison, the greedy rebalancer would not have degraded durability and availability, but would still

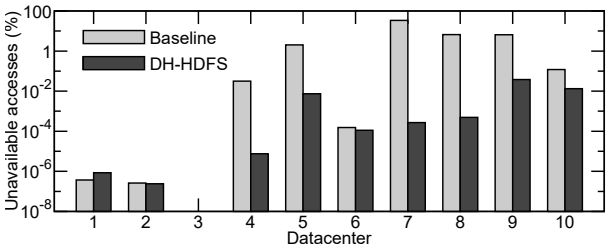


Figure 3: Data availability for baseline and DH-HDFS for 10 datacenters. The Y-axis is in log scale.

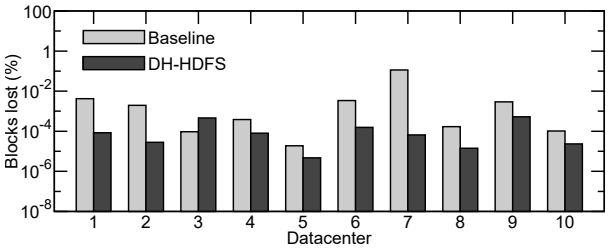


Figure 4: Data durability for baseline and DH-HDFS for 10 datacenters. The Y-axis is in log scale.

have moved 3.5× more data than our rebalancer.

Comparing baseline and DH-HDFS. Figure 3 quantifies the data availability (in percentage of failed accesses) of the baseline and DH-HDFS systems for our 10 real datacenters. The Y-axis in the figure is in log scale; a missing bar means that there were no failed accesses. To study a spectrum of utilization scenarios, we adjust the primary tenants' workloads (via acceleration/deceleration of their primary tenants' utilizations) to produce 3 groups of datacenters: the three leftmost datacenters exhibit low average primary tenant utilization (roughly 25% of the available resources), the four next datacenters exhibit mid-range average utilizations (roughly 50%), and the three rightmost datacenters exhibit high average utilizations (roughly 75%).

These results show that both systems exhibit negligible unavailability (> seven 9s availability) for the datacenters with low average utilization. For the mid-range datacenters, DH-HDFS improves availability by up to 5 orders of magnitude for three of them, while it matches the already high availability of the fourth (DC-6). The three high-utilization datacenters pose the greatest challenge to data availability. Still, DH-HDFS produces greater availability for all of them.

Figure 4 quantifies our datacenters' durability (in percentage of lost blocks) in the same order. Again, the Y-axis is in log scale. DH-HDFS exhibits greater durability than the baseline system by up to 4 orders of magnitude. The exception is DC-3 for which the baseline system produces slightly greater durability. The reason for this result is that consistent hashing provides statistical guarantees only. In exceptional cases, it may behave worse than assigning servers to subclusters by groups of primary tenants. We verified this by changing the hashing slightly to produce a different assignment, which makes our durability better than the baseline's.

Across all datacenters, our rebalancer migrates from 3.5× to 24× less data than the baseline's rebalancer.

Sensitivity of rebalancing to its parameters. Table 2 lists the comparisons we perform to assess the sensitivity of rebalancing to its main parameters. We show results for DC-7 (our largest production deployment), but other datacenters exhibit similar trends. Since durability and availability are not strongly affected by rebalancing (consistent hashing and Zhang's replica placement [26] are the dominant factors), we do not include these statistics. Instead, the table includes the range of subcluster peak loads ("Load range"), the range of subcluster space usage ("Space range"), and the amount of data migrated during rebalancing ("Data moved").

The first three rows of the table isolate the impact of the load threshold, assuming the other parameters are fixed at their default values and spare-based rebalancing is turned off. Looking at the load range and data moved columns, we can see that setting the load threshold at the average of the peak subcluster loads produces evenly balanced subclusters. Higher thresholds (including our default value of 40k accesses/second) produce more uneven peak loads, but can be satisfied with less data migration.

The next set of rows isolate the impact of the space threshold, assuming the other parameters stay at their default values and turning off load rebalancing. The space range and data moved columns show a similar effect: when the threshold is tight, rebalancing evens out the space usage at the cost of substantial data migration. Higher thresholds produce more unevenly distributed space usage, but involve less migration.

The last row shows the impact of using both average values for load- and space-driven rebalancing, assuming other parameters at their default values. This result shows that our full rebalancer brings both the peak load and space below their thresholds.

Finally, we study the impact of the frequency with which the rebalancer wakes up (not shown), while other parameters stay at their default values. We consider waking up every 30 minutes, 1 hour, and 2 hours. The results show that, for our setup, all these frequencies produce the same statistics as in the third row of the table.

6.3 Experimental results

We start this section by presenting experimental results on the performance of the DH-HDFS routers. We then study the performance of rebalancing operations.

Router performance. To explore the limits of our router's performance, we study two scenarios: a workload dominated by block reads, and a workload with

Study	Version	Load range	Space range	Data moved
Load threshold	average (30,500 acesses / sec)	27,250 - 30,600 accesses / sec	3 - 658 TB	33 TB
	35,000 accesses / sec	21,508 - 34,457 accesses / sec	6.5 - 665 TB	26 TB
	40,000 accesses / sec	21,508 - 37,834 accesses / sec	6.5 - 665 TB	24 TB
Space threshold	average (136 TB)	3,746 - 142,573 accesses / sec	122 - 132 TB	543 TB
	2 x average	5,873 - 141,789 accesses / sec	33 - 247 TB	439 TB
	4 x average	5,953 - 141,858 accesses / sec	16 - 495 TB	190 TB
Space and load	average, average	24,500 - 31,500 accesses / sec	117 - 136 TB	554 TB

Table 2: Rebalancing results for DH-HDFS and DC-7.

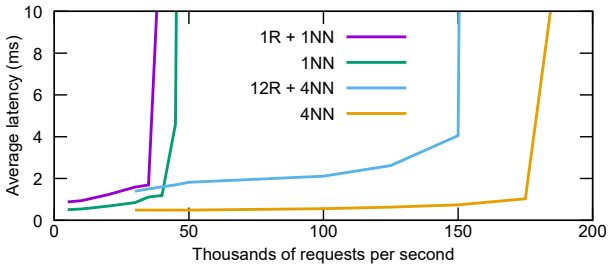


Figure 5: Performance for metadata operations.

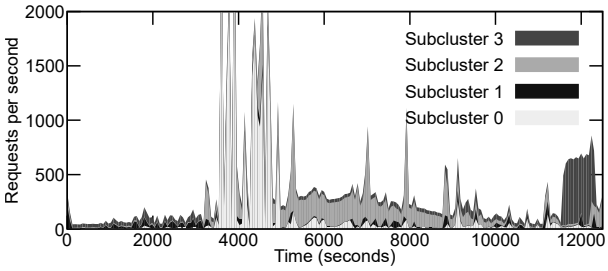


Figure 6: Subclusters' loads without rebalancing.

metadata-only operations. The former scenario is the best for our routers, and the latter is the worst; any real workload would perform between these extremes. In the best-case scenario, the performance of the routers is irrelevant. As each block is large (256MB), client-observed read latencies are measured in seconds (it takes seconds to read such a large block from disk), routers and NNs have little work to do (as clients contact DNs directly for blocks), and saturation occurs when the DNs saturate.

In the worst-case scenario, Figure 5 depicts the average metadata-only latency, as a function of load. This figure shows that one NN saturates at roughly 40k requests/second, whereas 4 NNs again saturate at roughly 4× higher load. In the small configuration, the routers add less than 1ms of latency and saturate slightly sooner. In the large configuration, the routers add up to 3ms of latency and saturate around 150k requests/second.

These results suggest that the routers perform well, adding relatively low latencies to metadata operations and negligible latencies to block accesses. Given that the latency of actual block transfers would likely dominate in real workloads, our routers should pose no significant overheads or bottlenecks in most scenarios.

Rebalancer performance. To explore the performance of rebalancing in our system, we study the Yahoo! trace when we replay it against a DH-HDFS setup with 4 subclusters and 4k servers. Figure 6 depicts the distribution of requests across the subclusters without rebalancing over time. The figure stacks the requests sent to each subcluster, representing them with different colors.

The figure shows that subcluster 0 receives a large amount of load around 4000 seconds into the execution. To demonstrate the rebalancer, we set the load watermark threshold at 2000 requests/second over any 5-minute period. As this threshold is exceeded, the rebalancer moves 4 folders of roughly the same size (400 files each) with a total of 13TB away from subcluster 0. Over repeated runs, we find that folder migrations take 354 seconds on average, 80% take less than 500 seconds, but one of them takes up to 25 minutes. Performance is especially variable when primary tenant traffic on the network is significant, *i.e.* during the weekdays. Most of the rebalancing time is spent in DistCP, with less than 34 seconds going into ensuring consistency and synchronizing the mount tables. The MILP solver takes negligible time (<100 milliseconds) to select the migrations.

These results demonstrate that the rebalancer itself is efficient, but the overall time to complete migrations can vary significantly, mainly due to primary tenant traffic. Nevertheless, recall that rebalances occur in the background and transparently to users, so the migration time variability is unlikely to be a problem.

File system performance. To illustrate the impact of the network traffic on the performance of our federated file system, Figure 7 shows Cumulative Distribution Functions (CDFs) of the performance of client-router interactions over the trace execution during a weekday (left) and during a weekend (right). The left graph shows much greater performance variability than the right one.

These results illustrate that harvesting spare resources for lower priority (secondary) workloads leaves their performance at the mercy of the primary tenants' resource demands. Most secondary workloads have lax performance requirements, so variability only becomes a problem when it is extreme. Nevertheless, if datacenter operators desire greater performance predictability for some of their secondary workloads, they must (1) account for these workloads in their resource provisioning, *e.g.* net-

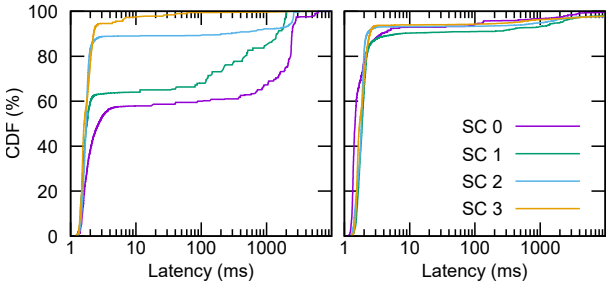


Figure 7: Client-router latency during weekday (left) and weekend (right).

work bandwidth; or (2) ensure that these workloads receive better than best-effort quality of service.

7 Lessons from production deployment

We deployed DH-HDFS in 4 production datacenters 6 months ago. The deployments currently involve more than 30k servers, and range from roughly 1k servers across 3 subclusters to more than 19k servers across 6 subclusters. We learned many lessons from these deployments and from onboarding users onto DH-HDFS.

Server-to-subcluster assignment and bootstrapping. Once we started deploying DH-HDFS, switching from manual server assignment to consistent hashing caused many servers to switch subclusters. This implied moving large amounts of data, which produced heavy network traffic and long downtimes. To avoid this data reshuffling, the administrators introduced a new service called the *Subcluster Controller*. This component maintained the server-to-subcluster assignments and authorized (or not) servers to join a subcluster. Servers with data from a subcluster are not allowed to join a different subcluster. Once a server is reimaged or decommissioned, the controller allows it to join the new subcluster assigned through consistent hashing.

File-to-subcluster assignment and onboarding users. Before introducing DH-HDFS, users submitted their batch workloads pointing to data of one subcluster (metadata manager). To onboard workloads gradually, we deployed the routers to listen to their own RPC and HTTP ports (instead of the metadata managers' ports).

Workloads that do not yet fully leverage the single DH-HDFS namespace still want to access subclusters directly. For this reason, we added special mount points that point to the root of each subcluster.

Spreading large datasets across subclusters. Even under DH-HDFS, workloads operating on large datasets were having difficulty (1) storing all their data in a single subcluster and (2) overloading the metadata manager. One option would have been to spread the files across folders in different subclusters, but users wanted this data in a single folder. For these users, we created special mount points that span multiple subclusters. Each file within such a mount point is assigned to one of the subclusters using consistent hashing. As explained in Section 4.3, this approach adds additional complexity for renaming. For this reason, we disallow renames and restrict these special mount points to certain workloads.

Rebalancing and administrators. Currently, the rebalancer is a service triggered by the administrator. It collects the space utilization and access statistics, and proposes which paths to move across subclusters. Our design expected paths to be unique across the namespace. However, administrators created multiple mount entries pointing to the same physical data (in the same subcluster). In this case, the federated namespace had loops and counted multiple times the same physical entity. In addition, we had the special mount points (*i.e.*, subcluster roots and folders spread across subclusters), which made the namespace even more complex. To handle these situations when collecting the statistics, we modified the rebalancer to (1) ignore the special mount points; and (2) map all aliases to a single federated location. For example, if `/tmp/logs` and `/logs` both point to `/logs` in subcluster 0, we assign all the accesses to just one path.

Performance in production. Our largest deployment has 24 routers for 6 subclusters, and typically runs large data analytics workloads on an index of the Web. The load across the routers and their latency are fairly even. The latency of the routers is around 3 milliseconds, whereas the latency of the metadata managers is around 1 millisecond. These match the latencies from Section 6.

For this deployment, we use a 5-server Zookeeper ensemble for the state store. On average, a router sends 5 requests to the store every 10 seconds. This is a low load compared to the other services that use the ensemble.

8 Conclusions

In this paper, we proposed techniques for automatically and transparently scaling the distributed file systems used in commercial datacenters. We focused on systems where interactive services and batch workloads share the same servers, but most of our work also applies to dedicated servers. Our results show that our techniques introduce little overhead, and our system behaves well even in extreme scenarios. We conclude that it is possible to scale existing systems to very large sizes in a simple and efficient manner, while exposing a single namespace.

Acknowledgments

We thank the ATC reviewers and our shepherd, Vishakha Gupta, for their comments on our paper. We also thank Bing's Multitenancy team, Chris Douglas, Carlo Curino, and John Douceur for many suggestions and discussions, their comments on our paper, and help open-sourcing our system and deploying it in production.

References

[1] 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. In *OSDI* (2002).

[2] APACHE FOUNDATION. HDFS Architecture Guide, 2008. `http://hadoop.apache.org/docs/current/hdfs_design.html`.

[3] APACHE FOUNDATION. ViewFs Guide, 2016. `http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/ViewFs.html`.

[4] ARENE, M., CHALIPARAMBIL, K., CURINO, C., DOUGLAS, C., FUMAROLA, G. M., HEDDAYA, S., KRISHNAN, S., RAMAKRISHNAN, R., RAO, S., SAKALANAGA, S., SHAH, R., SHI, B., AND ZHOU, B. Enable YARN RM Scale Out via Federation using Multiple RM's. `https://issues.apache.org/jira/browse/YARN-2915`.

[5] BARROSO, L. A., CLIDARAS, J., AND HÖLZLE, U. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool Publishers, 2009.

[6] BEAVER, D., KUMAR, S., LI, H. C., SOBEL, J., VAJGEL, P., ET AL. Finding a Needle in Haystack: Facebook's Photo Storage. In *OSDI* (2010).

[7] CHAIKEN, R., JENKINS, B., LARSON, P. Å., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. In *VLDB* (2008).

[8] 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).

[9] DELIMITROU, C., AND KOZYRAKIS, C. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *ASPLOS* (2014).

[10] DOUCEUR, J. R., AND HOWELL, J. Distributed Directory Service in the Farsite File System. In *OSDI* (2006).

[11] FOUNDATION, A. HDFS Federation, 2016. `https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-hdfs/Federation.html`.

[12] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S. T. The Google File System. In *SOSP* (2003).

[13] HILDRUM, K., DOUGLIS, F., WOLF, J. L., YU, P. S., FLEISCHER, L., AND KATTA, A. Storage Optimization for Large-Scale Distributed Stream-Processing Systems. *Transactions on Storage (TOS) 3*, 4 (2008), 5.

[14] HSIAO, H. C., CHUNG, H. Y., SHEN, H., AND CHAO, Y. C. Load Rebalancing for Distributed File Systems in Clouds. *TPDS 24*, 5 (2013), 951–962.

[15] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX ATC* (2010).

[16] KACE, J., AND GOIRI, I. Router-based HDFS federation, 2017. `https://issues.apache.org/jira/browse/HDFS-10467`.

[17] KARGER, D., LEHMAN, E., LEIGHTON, T., PANIGRAHY, R., LEVINE, M., AND LEWIN, D. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *STOC* (1997).

[18] LO, D., CHENG, L., GOVINDARAJU, R., RANGANATHAN, P., AND KOZYRAKIS, C. Heracles: Improving Resource Efficiency at Scale. In *ISCA* (2015).

[19] 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 *MICRO* (2011).

[20] MUTHITACHAROEN, A., MORRIS, R., GIL, T. M., AND CHEN, B. Ivy: A Read/Write Peer-to-Peer File System. In *OSDI* (2002).

[21] SINGH, A., KORUPOLU, M., AND MOHAPATRA, D. Server-Storage Virtualization: Integration and Load Balancing in Data Centers. In *Supercomputing* (2008).

[22] VERMA, A., PEDROSA, L., KORUPOLU, M., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-Scale Cluster Management at Google with Borg. In *EuroSys* (2015).

[23] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D., AND MALTZAHN, C. Ceph: A Scalable, High-Performance Distributed File System. In *OSDI* (2006).

[24] WILKES, J. Private communication, 2016.

[25] YAHOO! Yahoo! Research Webscope Program, 2008. https://webscope.sandbox.yahoo.com/.

[26] 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 *OSDI* (2016).