conference
proceedings

2012 USENIX Annual Technical Conference (USENIX ATC '12)

Boston, MA, USA
June 13–15, 2012

Sponsored by
usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Proceedings of the 2012 USENIX Annual Technical Conference
Boston MA, USA June 13–15, 2012

ISBN 978-931971-93-5

USENIX Association

Proceedings of the
2012 USENIX Annual Technical Conference
(USENIX ATC '12)

June 13–15, 2012
Boston MA, USA

Conference Organizers

Program Co-Chairs

Gernot Heiser, *NICTA and University of New South Wales*
Wilson Hsieh, *Google Inc.*

Program Committee

James Anderson, *University of North Carolina, Chapel Hill*
Katerina Argyraki, *École Polytechnique Fédérale de Lausanne (EPFL)*
Andrea Arpaci-Dusseau, *University of Wisconsin—Madison*
Andrew Birrell, *Microsoft Research*
Pei Cao, *Facebook*
John Carter, *IBM Research*
Rohit Chandra, *Yahoo!*
Haibo Chen, *Shanghai Jiao Tong University*
Frank Dabek, *Google*
Angela Demke Brown, *University of Toronto*
Alexandra Fedorova, *Simon Fraser University*
Andreas Haeberlen, *University of Pennsylvania*
Jon Howell, *Microsoft Research*
Sam King, *University of Illinois at Urbana-Champaign*
Eddie Kohler, *Harvard University*
Donald Kossmann, *ETH Zurich*
Hank Levy, *University of Washington*
Pradeep Padala, *VMware*
John Regehr, *University of Utah*
Mahadev Satyanarayanan, *Carnegie Mellon University*
Wolfgang Schröder-Preikschat, *Friedrich-Alexander University Erlangen-Nuremberg*
Liuba Shrira, *Brandeis University*
Emil Sit, *Hadapt*
Christopher Small, *NetApp*
Udo Steinberg, *Intel*
Emmett Witchel, *University of Texas, Austin*

Poster Session Chair

Emil Sit, *Hadapt*

The USENIX Association Staff

External Reviewers

David G. Andersen
Andrea Bastoni
Andrew Baumann
Frank Bellosa
Tom Bergen
Jeff Bigham
George Bissias
Richard Black
Marco Canini
David Cock
Tobias Distler
Mihai Dobrescu
Alan M. Dunn
Glenn Elliott
Anja Feldmann
James Fogarty
Bryan Ford
Felix Freiling
Moises Goldszmidt
Vishakha Gupta
Werner Haas
Wanja Hofer
Anne Holler
Timo Hönig
Adam Hupp
Tomas Isdal
Shinpei Kato
Christopher Kenna
Karl Koscher
Arvind Krishnamurthy
Srinivas Krishnan
Charles Lamb
Stefan Leue
Alex Lloyd
Fabian Monrose
Toby Murray
Arjun Narayan
Hugo Patterson
Franz J. Rammig
Charlie Reis
Luigi Rizzo
Chris Rossbach
Leonid Ryzhyk
Fabian Scheler
Simon Schubert
Will Scott
Micah Sherr
Arrvindh Shriraman
Mark S. Silberstein
Michael Sirivianos
Maxim Smith
Julian Stecklina
Isabella Stilkerich
Mike Swift
Reinhard Tartler
Mustafa Uysal
Matthew Wachs
Quanqing Xu
Jie Yu
Cristian Zamfir
Mingchen Zhao

2012 USENIX Annual Technical Conference
June 13–15, 2012
Boston, MA, USA

Message from the Program Co-Chairs. vii

Wednesday, June 13

9:00–10:30 Cloud

Demand Based Hierarchical QoS Using Storage Resource Pools 1
Ajay Gulati and Ganesha Shanmuganathan, VMware Inc.; Xuechen Zhang, Wayne State University; Peter Varman, Rice University

Erasure Coding in Windows Azure Storage 15
Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin, Microsoft Corporation

Composable Reliability for Asynchronous Systems 27
Sunghwan Yoo, Purdue University and HP Labs; Charles Killian, Purdue University; Terence Kelly, HP Labs; Hyoun Kyu Cho, HP Labs and University of Michigan; Steven Plite, Purdue University

11:00–12:30 Multicore

Managing Large Graphs on Multi-Cores with Graph Awareness 41
Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haridasan, Microsoft Research

MemProf: A Memory Profiler for NUMA Multicore Systems 53
Renaud Lachaize, UJF; Baptiste Lepers, CNRS; Vivien Quéma, GrenobleINP

Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications 65
Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller, LIP6/INRIA

1:30–3:30 Packet Processing

The Click2NetFPGA Toolchain 77
Teemu Rinta-aho and Mika Karlstedt, NomadicLab, Ericsson Research; Madhav P. Desai, Indian Institute of Technology (Bombay)

Building a Power-Proportional Software Router 89
Luca Niccolini, University of Pisa; Gianluca Iannaccone, RedBow Labs; Sylvia Ratnasamy, University of California, Berkeley; Jaideep Chandrashekar, Technicolor Labs; Luigi Rizzo, University of Pisa and University of California, Berkeley

netmap: A Novel Framework for Fast Packet I/O. 101
Luigi Rizzo, Università di Pisa, Italy

Toward Efficient Querying of Compressed Network Payloads 113
Teryl Taylor, UNC Chapel Hill; Scott E. Coull, RedJack; Fabian Monrose, UNC Chapel Hill; John McHugh, RedJack

Thursday, June 14

8:30–10:30 Security

Body Armor for Binaries: Preventing Buffer Overflows Without Recompilation 125
Asia Slowinska, Vrije Universiteit Amsterdam; Traian Stancescu, Google, Inc.; Herbert Bos, Vrije Universiteit Amsterdam

Abstractions for Usable Information Flow Control in Aeolus 139
Winnie Cheng, IBM Research; Dan R.K. Ports and David Schultz, MIT CSAIL; Victoria Popic, Stanford; Aaron Blankstein, Princeton; James Cowling and Dorothy Curtis, MIT CSAIL; Liuba Shrira, Brandeis; Barbara Liskov, MIT CSAIL

Treehouse: Javascript Sandboxes to Help Web Developers Help Themselves 153
Lon Ingram, The University of Texas at Austin and Waterfall Mobile; Michael Walfish, The University of Texas at Austin

Cloud Terminal: Secure Access to Sensitive Applications from Untrusted Systems 165
Lorenzo Martignoni, University of California, Berkeley; Pongsin Poosankam, University of California, Berkeley, and Carnegie Mellon University; Matei Zaharia, University of California, Berkeley; Jun Han, Carnegie Mellon University; Stephen McCamant, Dawn Song, and Vern Paxson, University of California, Berkeley; Adrian Perrig, Carnegie Mellon University; Scott Shenker and Ion Stoica, University of California, Berkeley

11:00–Noon Short Papers: Tools and Networking

Mosh: An Interactive Remote Shell for Mobile Clients 177
Keith Winstein and Hari Balakrishnan, M.I.T. Computer Science and Artificial Intelligence Laboratory

TROPIC: Transactional Resource Orchestration Platform in the Cloud 183
Changbin Liu, University of Pennsylvania; Yun Mao, Xu Chen, and Mary F. Fernández, AT&T Labs—Research; Boon Thau Loo, University of Pennsylvania; Jacobus E. Van der Merwe, AT&T Labs—Research

Trickle: Rate Limiting YouTube Video Streaming 191
Monia Ghobadi, University of Toronto; Yuchung Cheng, Ankur Jain, and Matt Mathis, Google

Tolerating Overload Attacks Against Packet Capturing Systems 197
Antonis Papadogiannakis, FORTH-ICS; Michalis Polychronakis, Columbia University; Evangelos P. Markatos, FORTH-ICS

Enforcing Murphy's Law for Advance Identification of Run-time Failures 203
Zach Miller, Todd Tannenbaum, and Ben Liblit, University of Wisconsin—Madison

1:30–3:30 Distributed Systems

A Scalable Server for 3D Metaverses 209
Ewen Cheslack-Postava, Tahir Azim, Behram F.T. Mistree, and Daniel Reiter Horn, Stanford University; Jeff Terrace, Princeton University; Philip Levis, Stanford University; Michael J. Freedman, Princeton University

Granola: Low-Overhead Distributed Transaction Coordination 223
James Cowling and Barbara Liskov, MIT CSAIL

High Performance Vehicular Connectivity with Opportunistic Erasure Coding 237
Ratul Mahajan, Jitendra Padhye, Sharad Agarwal, and Brian Zill, Microsoft Research

Server-assisted Latency Management for Wide-area Distributed Systems 249
Wonho Kim, Princeton University; KyoungSoo Park, KAIST; Vivek S. Pai, Princeton University

Thursday, June 14 (continued)

4:00–5:30 Deduplication

Generating Realistic Datasets for Deduplication Analysis 261
Vasily Tarasov and Amar Mudrankit, Stony Brook University; Will Buik, Harvey Mudd College; Philip Shilane, EMC Corporation; Geoff Kuenning, Harvey Mudd College; Erez Zadok, Stony Brook University

An Empirical Study of Memory Sharing in Virtual Machines 273
Sean Barker, University of Massachusetts Amherst; Timothy Wood, The George Washington University; Prashant Shenoy and Ramesh Sitaraman, University of Massachusetts Amherst

Primary Data Deduplication—Large Scale Study and System Design. 285
Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Oltean, Jin Li, and Sudipta Sengupta, Microsoft Corporation

Friday, June 15

8:30–10:30 Languages and Tools

Design and Implementation of an Embedded Python Run-Time System 297
Thomas W. Barr, Rebecca Smith, and Scott Rixner, Rice University

AddressSanitizer: A Fast Address Sanity Checker 309
Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov, Google

Software Persistent Memory 319
Jorge Guerra, Leonardo Mármol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei, Florida International University

Rivet: Browser-agnostic Remote Debugging for Web Applications 333
James Mickens, Microsoft Research

11:00–Noon Short Papers: Performance

Wimpy Nodes with 10GbE: Leveraging One-Sided Operations in Soft-RDMA to Boost Memcached. 347
Patrick Stuedi, Animesh Trivedi, and Bernard Metzler, IBM Research, Zurich

Revisiting Software Zero-Copy for Web-caching Applications with Twin Memory Allocation 355
Xiang Song and Jicheng Shi, Shanghai Jiao Tong University and Fudan University; Haibo Chen, Shanghai Jiao Tong University; Binyu Zang, Shanghai Jiao Tong University and Fudan University

Seagull: Intelligent Cloud Bursting for Enterprise Applications. 361
Tian Guo and Upendra Sharma, UMASS Amherst; Timothy Wood, The George Washington University; Sambit Sahu, IBM Watson; Prashant Shenoy, UMASS Amherst

The Forgotten 'Uncore': On the Energy-Efficiency of Heterogeneous Cores 367
Vishal Gupta, Georgia Tech; Paul Brett, David Koufaty, Dheeraj Reddy, and Scott Hahn, Intel Labs; Karsten Schwan, Georgia Tech; Ganapati Srinivasa, Intel Corporation

1:00–2:30 OS

Software Techniques for Avoiding Hardware Virtualization Exits. 373
Ole Agesen, Jim Mattson, Radu Rugina, and Jeffrey Sheldon, VMware

AppScope: Application Energy Metering Framework for Android Smartphone Using Kernel Activity Monitoring 387
Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha, Yonsei University, Korea

Gdev: First-Class GPU Resource Management in the Operating System. 401
Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt, UC Santa Cruz

Friday, June 15 (continued)

3:00–5:00 Replication

Gnothi: Separating Data and Metadata for Efficient and Available Storage Replication 413
Yang Wang, Lorenzo Alvisi, and Mike Dahlin, The University of Texas at Austin

Dynamic Reconfiguration of Primary/Backup Clusters 425
Alexander Shraer and Benjamin Reed, Yahoo! Research; Dahlia Malkhi, Microsoft Research; Flavio Junqueira, Yahoo! Research

Surviving Congestion in Geo-Distributed Storage Systems 439
Brian Cho, University of Illinois at Urbana-Champaign; Marcos K. Aguilera, Microsoft Research Silicon Valley

Practical Hardening of Crash-Tolerant Systems. 453
Miguel Correia, IST-UTL/INESC-ID; Daniel Gómez Ferro, Flavio P. Junqueira, and Marco Serafini, Yahoo! Research

Message from the 2012 USENIX Annual Technical Conference Program Co-Chairs

Welcome to the 2012 USENIX Annual Technical Conference!

This year's conference demonstrates that there is a huge amount of interest in USENIX ATC. We received 185 full-paper and 45 short-paper submissions, which is a further 30% increase over the submission record set at last year's conference. Authors were from institutions in over a dozen countries spread over four continents: a truly international community.

From these submissions, the program committee selected 32 full papers and 9 short papers for the conference program. These papers cover the spectrum of systems work, with cloud computing a focus area. We have followed the ATC tradition of publishing both ground-breaking research and practically oriented tools and experience papers. We had hoped to get some papers on experimentally verifying prior results or designs, but that kind of paper is still outside the norm of systems conferences. Maybe next year!

We had a great program committee: a total of 27 members, 11 of whom are from industry and 16 from academic institutions. Program committee members were allowed to submit papers, which were held to a higher standard; the co-chairs were not associated with any submissions. We deeply appreciate the committee's diligence in reviewing papers: 705 reviews in total, for an average of 26 reviews per PC member. We also asked 61 external reviewers to write an additional 69 reviews when it was necessary to get some outside expertise. The review process wound up being three rounds: if no paper had a champion after any round, it did not survive into the next round (although PC members were allowed to initiate discussion at the meeting on any paper). Each paper received two reviews in the first round, and could have received a third review in the second round, and a fourth and possibly fifth review in the third round. The committee met at the Google office in New York City on March 21, 2012, for an all-day meeting; three members had to videoconference into the meeting. After the meeting, every paper was shepherded. We would like to acknowledge the integrity of the authors of two papers that were withdrawn during the shepherding process because those authors could not reproduce their results.

Besides the paper authors and reviewers, who define the research community for USENIX ATC, there are a lot of other people and organizations that deserve mention. First, the USENIX staff has been wonderful to work with; without their support, our jobs would have been much harder. They made it possible for us to focus on getting the conference program defined. Second, Eddie Kohler's HotCRP software was indispensable in running the PC. Third, we thank Google for hosting and helping to sponsor the meeting. Finally, we thank our corporate sponsors: Gold Sponsors Facebook and VMware; Silver Sponsors Google, EMC, and Microsoft Research; and Bronze Sponsor NetApp.

Thank you for participating in the USENIX ATC community, and enjoy the conference!

Gernot Heiser, ***NICTA and University of New South Wales***
Wilson Hsieh, ***Google***

Demand Based Hierarchical QoS Using Storage Resource Pools

Ajay Gulati, Ganesha Shanmuganathan
VMware Inc
{*agulati, sganesh*}*@vmware.com*

Xuechen Zhang
Wayne State University
xczhang@wayne.edu

Peter Varman
Rice University
pjv@rice.edu

Abstract

The high degree of storage consolidation in modern virtualized datacenters requires flexible and efficient ways to allocate IO resources among virtual machines (VMs). Existing IO resource management techniques have two main deficiencies: (1) they are restricted in their ability to allocate resources across multiple hosts sharing a storage device, and (2) they do not permit the administrator to set allocations for a group of VMs that are providing a single service or belong to the same application.

In this paper we present the design and implementation of a novel software system called *Storage Resource Pools* (SRP). SRP supports the logical grouping of related VMs into hierarchical pools. SRP allows reservations, limits and proportional shares, at both the VM and pool levels. Spare resources are allocated to VMs in the same pool in preference to other VMs. The VMs may be distributed across multiple physical hosts without consideration of their logical groupings. We have implemented a prototype of storage resource pools in the VMware ESX hypervisor. Our results demonstrate that SRP provides hierarchical performance isolation and sharing among groups of VMs running across multiple hosts, while maintaining high utilization of the storage device.

1 Introduction

Shared storage access and data consolidation is on the rise in virtualized environments due to its many benefits: universal access to data, ease of management, and support for live migrations of virtual machines (VMs). Multi-tiered SSD-based storage devices, with high IO rates, are driving systems towards ever-higher consolidation ratios. A typical virtualized cluster consists of tens of servers, hosting hundreds of VMs running diverse applications, and accessing shared SAN or NAS based storage devices.

To maintain control over workload performance, storage administrators usually deploy separate storage devices (also called as LUNs or datastores) for applications requiring strong performance guarantees. This approach has several drawbacks: growing LUN sprawl, higher management costs, and over-provisioning due to reduced benefits from multiplexing. Encouraging LUN sharing among diverse clients requires systems to provide better controls to isolate the workloads and enable QoS differentiation. Recently, PARDA [6] and mClock [8] have been proposed to provide storage QoS support. However, these and other existing approaches like SFQ(D) [12], Triage [13] , Façade [15], Zygaria [24], pClock [7] etc. either provide only proportional allocation or require a centralized scheduler (see Section 2.2).

In this paper, we present a new software system called **storage resource pools** (SRPs) with the following desirable properties:

Rich Controls: QoS can be specified using throughput *reservations* (lower bounds), *limits* (upper bounds) and *shares* (proportional sharing). These may be set for individual VMs or collectively for a group of related VMs known as a resource pool. Reservations are absolute guarantees, that specify the minimum amount of service that a VM (or group) must receive. Limits specify the maximum allocation that should be made to the VM or the group. These are useful for enforcing strict isolation and restricting tenants to contractually-set IOPS based on their SLOs. Shares provide a measure of relative importance between VMs or groups, and are used for proportional allocation when capacity is constrained.

Hierarchical Pooling: Storage administrators can define *storage resource pools* (SRPs) to logically partition IO resources in a hierarchical manner. SRPs allow related VMs to be treated as a single unit for resource allocation. These units can be aggregated into larger SRPs to create a resource pool hierarchy. Resource pooling has several advantages; it (1) spares the user from having to set per-VM controls that are hard to determine; (2) allocates resources to divisions or departments based on organizational structure; and (3) allocates resources to a

group of VMs that are working together to provide a single service. The latter scenario is becoming increasingly common with dynamic websites like e-Commerce and social-networking, where a webpage may be constructed by involving several virtual machines.

Dynamic Allocation based on Demand: SRPs can *dynamically* reallocate LUN capacity (IOPS) among VMs based on the current workload demands, while respecting user-set constraints (see Section 2).

Distributed and Scalable Operation: VMs comprising a resource pool may be distributed across multiple servers (hosts), and a single server may run VMs belonging to many different resource pools. Such distributed architectures are very common in virtualized datacenters.

Providing these controls is quite challenging for several reasons: (1) VMs in the same pool may be distributed across multiple hosts; (2) there is no central location to implement an IO scheduler that sees the requests from all the hosts; and (3) workload demands and device IOPS are highly variable and need to be tracked periodically for an effective implementation of resource pooling.

We have implemented a prototype of storage resource pools on the VMware ESX Server hypervisor [19]. In our prototype, an administrator can create one resource pool per storage device. Our extensive evaluation with multiple devices and workloads shows that SRPs are able to provide the desired isolation and aggregation of IO resources across various VM groups, and dynamically adapt allocation to the current VM demands.

The rest of the paper is organized as follows. Section 2 presents an example to motivate the need for storage resource pools and discusses related work in this area. Section 3 presents the SRP design in detail. In Section 4 we discuss implementation details and storage-specific issues. Section 5 presents the results of extensive performance evaluation that demonstrates the power and effectiveness of storage resource pools. Finally we conclude with some directions for future work in Section 6.

2 Motivation and Related Work

In this section we first motivate the need for storage resource pools using a simple example and discuss the challenges in implementing them in a distributed cluster. We then review the literature on IO resource management and the limitations of existing QoS techniques.

2.1 Need For Storage Resource Pools

Consider an enterprise that has virtualized its infrastructure and consolidated its IO workloads on a small set of storage devices. VMs from several different divisions, (say sales and finance for example), may be deployed on the same device (also called as datastore). The administrator sets up a pool for the divisions with settings reflecting the importance of their workloads. The VMs of the sales division (handling sales in different continents) may need an overall reservation of 1000 IOPS. This total reservation is flexibly shared by these VMs based on the peaks and troughs of demand in different time zones. The finance division is running background data analytics in their VMs and the administrator wants to restrict their combined throughput to 500 IOPS, to reduce their impact on critical sales VMs.

In addition to reservation and limit controls, the administrator may want the VMs from the sales division to get more of the spare capacity (*i.e.*capacity left after satisfying reservations) than VMs from the finance division. This is only relevant during contention periods when demand is higher than the current capacity. For this, one can set shares at the resource pool level and the allocation is done in proportion to the share values. Shares can also be used to do prioritized allocation among the VMs within the same pool.

Many of these requirements can be met by using hard VM-level settings. However, that will not allow the IOPS to be dynamically shared among VMs of a group based on demand, as needed by the sales division. Similarly, one will have to individually limit each VM, which is more restrictive than setting the limit on a group of them. Moreover if a VM gets idle, the resource should flow first to the VMs of the same group rather than to a different group.

Figure 1(a) shows a storage resource pool with two children Sales and Finance. These child nodes have reservation (R) and limit (L) settings as per the company policy. Both of them also have two child VMs with settings as shown. The reservation for the Sales node (1000) is higher than the sum of the reservations (300) of its child VMs; the excess amount will be dynamically allocated to the children to increase their statically-set reservations (r), based on their current demand and other settings. Similarly, in the case of the Finance node, the parent limit (500) is less than the sum of the limits (1000) of the individual child VMs; hence, we need to dynamically set the limit on the two child VMs to sum to the parent's value. Once again the allocation is made dynamically based on the current distribution of demand among these VMs, and the other resource control settings.

Also note that we have assigned twice the number of shares to the Sales pool, which means that the capacity at the root will be allocated among the Sales and Finance pools in the ratio 2:1, unless that would lead to violating the reservation or limit settings at the node. We have also allocated different shares to VMs in the Sales pool to allow them to get differentiated service during periods of contention.

Technique	*Distributed*	*Reservation*	*Limit*	*Share*	*Hierarchical*	Storage Allocation
Proportional sharing techniques	No	No	No	Yes	No	Yes
Distributed mechanisms (PARDA [6])	Yes	No	No	Yes	No	Yes
Centralized IO schedulers	No	Yes	Yes	Yes	Some	Yes
ESX CPU scheduler	No	Yes	Yes	Yes	Yes	No
ESX Memory scheduler	No	Yes	Yes	Yes	Yes	No
Storage Resource Pools	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**

Table 1: Comparison of storage resource pools with existing resource allocation schemes

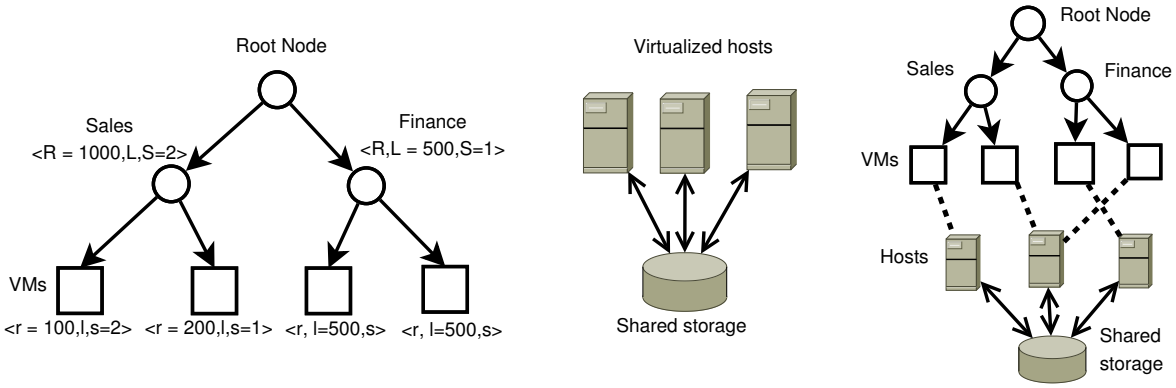


Figure 1: Storage resource pools description and mapping to physical resources

This task of enforcing the desired controls in SRP is challenging because the storage device is accessed by multiple hosts in a distributed manner using a clustered file system like VMFS [1] (in our case) or NFS, with no centralized control on the IO path as shown in Figure 1(b). Finally, based on the requirements for other resources such as CPU and memory, the VMs may get dynamically placed or moved among hosts using live migration. Thus, the system should adapt to the dynamic placement of VMs and cannot rely on static settings. Figure 1(c) shows an example mapping of the VMs to hosts.

Resource Pool Semantics: In summary, the resource pool semantics dictate the following allocation at each level of the resource pool (RP) tree: (1) Distribute the parent's reservation among its children in proportion to their shares while ensuring that each child gets at least its own reservation and no more than its demand or static limit; (2) Distribute the parent's limit among its children in proportion to their shares while making sure that no child gets more than its own static limit or demand; and (3) Distribute the parent's share to its children in proportion to their shares.

2.2 Previous Work

We classified existing work on QoS controls for storage into three categories, as discussed below. Table 1 provides a summary of existing approaches and their comparison with Storage Resource Pools.

Proportional Sharing: Many approaches such as Stonehenge [10], SFQ(D) [12] have been proposed for proportional or weighted allocation of IO resources. These techniques are based on fair-queuing algorithms proposed for network bandwidth allocation (WFQ [3], SFQ [5]) but they deploy storage-specific optimizations to maintain higher efficiency of the underlying storage system. DSFQ [23] proposed proportional allocation for distributed storage, but it needs specific cooperation between the underlying storage device and storage clients.

Different from throughput allocation, Argon [21] and Fahrrad [16] proposed time-sliced allocation of disk accesses to reduce interference across multiple streams accessing the device. Façade [15] presented a combination of EDF based scheduling and queue depth manipulation to provide SLOs to each workload in terms of IOPS and latency. The reduction in queue depth to meet latency bounds can have severe impact on the overall efficiency of the underlying device. SARC+Avatar [25] improved upon that concern by providing better bounds on the queue depth and a trade-off between throughput and latency.

Unlike these centralized schedulers, PARDA [6] provided a distributed proportional-share algorithm to allocate LUN capacity to VMs running on different hosts.

PARDA also runs across a cluster of ESX hosts, but it doesn't support reservation and limit controls. A limitation of pure share-based allocation is that it cannot guarantee a lower bound on absolute VM throughput. Consequently, VMs with strict QoS requirements suffer when the aggregate IO rate of the LUN drops or if new VMs are added on the same LUN. In addition, PARDA does not support resource pooling so VMs running on different hosts are completely independent.

Triage [13] uses a centralized control mechanism that creates an adaptive model of the storage system and sets per-client bandwidth caps to allocate a specific share of the available capacity. Doing per-client throttling using bandwidth caps can underutilize array resources if the workloads become idle. Triage also doesn't support resource pooling unlike SRP.

Algorithms with Reservation Support: The problem of resource reservations for CPU, memory and storage management are well studied. Several approaches [4, 17] have been proposed to support CPU reservations for real-time applications while maintaining proportional resource sharing. Since CPU capacity is fixed and not significantly affected by workloads, it is relatively straightforward to provide CPU reservation in MHz. The VMware ESX server [20, 22] has been providing reservation, limit and shares based allocation to VMs for both CPU and memory since 2003.

For the allocation of storage resources, mClock [8] proposed a per-host local scheduler that provides all three controls (reservation, limit and shares) for VMs running on a *single host*. mClock does this by using three separate tags per client, one for each of the controls. The tags are assigned using real-time instead of virtual time and the scheduler dynamically switches between the tags for scheduling. However, in a clustered environment, a host-level algorithm alone cannot control the LUN capacity available to a specific host due to workload variations on other hosts in the cluster. Hence, *any solution local to a single host* is unable to provide guarantees across a cluster and is not sufficient for our use case.

Hierarchical Resource Management: CPU and memory resource pools [20] implemented by the VMware ESX server since 2003, were proposed for hierarchical resource management. However, the existing solutions were not designed for storage devices which are stateful and have fluctuating capacity. More importantly, both CPU and memory are local to a host and a centralized algorithm suffices to do resource allocation. Zygaria [24] proposed a hierarchical token-bucket based centralized IO scheduler to provide hierarchical resource allocation while supporting reservations, limits and a statistical notion of shares.

Storage resource pools, by contrast, need to work across a cluster of hosts that are accessing a storage device in a distributed manner. This makes it harder to use any centralized IO scheduler. Furthermore, earlier approaches use a fixed queue depth for the underlying device, which is hard to determine in practice; SRP varies the queue depth in order to ensure high device utilization. Finally, SRP adjusts VM-level controls adaptively based on the demand, so a user does not have to specify all of the per-VM settings.

3 Storage Resource Pool Design

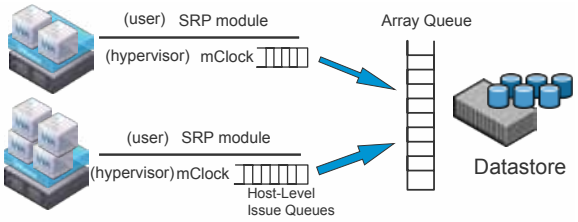


Figure 2: SRP system architecture

In this section, we discuss the key concepts and overall design of storage resource pools. Figure 2 shows the overall system architecture with multiple virtualized hosts accessing a shared storage array. Each host has two main components: an **SRP module** and a local IO scheduler **mClock** [8], that synergistically control the allocations to the VMs. The SRP module is responsible for determining how much of the array capacity should be provided to each VM and each host. mClock is responsible for scheduling the VMs on each host in accordance with the allocations.

The SRP module is a user-level process that runs directly on the ESX hypervisor. Each SRP module periodically decides how much of the array capacity to allocate to this host and the VMs running on it for the next interval. The amount is based on several factors, both static and dynamic: the structure of the RP tree, the control settings (static R,L,S) on the nodes, the dynamic demand of the VMs, the dynamic array capacity for the current workload mix, and the mapping of the VMs to hosts.

SRP computes two quantities periodically: (1) the dynamic VM settings (R,L,S) in accordance with the global resource pool constraints and VM demands, and (2) the *issue queue depth* per host. The maximum number of requests that a host can keep outstanding at the array is bounded by the issue queue depth (also called **host queue depth**), as shown in Figure 2. The size of the issue queue reflects the array capacity currently allocated to the host. Next we discuss the functionality of SRP Module in more detail.

3.1 SRP Module

Algorithm 1 provides a high-level description of the SRP module. It performs three major tasks: 1) updates demand on the RP-tree nodes; 2) computes new values for the reservations, limits, and shares for the VMs for use by the mClock scheduler – these are also called dynamic R,L,S values respectively; and 3) estimates the new array capacity and divides it among the hosts. The inputs to the module are the statistics collected during the previous monitoring interval, specifically the VM demands and measured latency.

Algorithm 1: SRP Module

/* Run periodically to reallocate IO resources */

1. **Update demand in RP Tree**
   (*a*) Update demand of VMs in RP tree
   (*b*) Update demands at internal RP-tree nodes by aggregating demands of its children
2. **Compute dynamic R,L,S**
   (*a*) Update VM Reservation (R Divvy)
   (*b*) Update VM Limits (L Divvy)
   (*c*) Update VM Shares (S Divvy)
3. **Update Array and Host Queue Depths**
   (*a*) Estimate new *array queue depth*
   (*b*) Compute VM entitlement
   (*c*) Set *host queue depth*

3.1.1 Update and Aggregate Demand

The ESX hypervisor maintains stats on the aggregated latency and total number of IOs performed by each VM. Using these stats, the SRP module determines the average latency (*avgLatency*) and average IOPS (*avgIops*), and computes the **VM demand** in terms of the average number of outstanding IOs (*demandOIO*) using Little's law [14] (see equation 1). Each SRP module owns a block in a shared file on the underlying datastore and updates the VM-level stats in that file. By reading this file, every host can get the VM demand values in terms of outstanding IOs (also called as OIOs). The SRP module on each host then converts the *demandOIO* (see equation 2) to a normalized demand IOPS value (*demandIops*), based on the storage device congestion threshold latency ($\mathscr{L}_c$). This helps to avoid overestimating a VM's demand based on local latency variations.

$$demandOIO = avgLatency \times avgIops \quad (1)$$

$$demandIops = demandOIO/\mathscr{L}_c \quad (2)$$

$$demandIops = min(max(demandIops, R), L) \quad (3)$$

The congestion threshold is the maximum latency at which the storage device is operated. This ensures a high utilization of the underlying device. Based on our experiments we have found the range between 20 to 30 *ms* to be good enough for disk-based storage devices. For SSD-backed LUNs, $\mathscr{L}_c$ can be set to a lower value (*e.g.* 5 ms). Our implementation is not sensitive to this control but the utilization of the underlying device may get impacted. The SRP module controls the array queue depth to keep the latency close to $\mathscr{L}_c$, so that we utilize the device in an efficient manner.

The *demandIops* value is then adjusted to make sure that it lies within the lower and upper bounds represented by the reservation and limit settings for each VM (see equation 3). Finally the demand is aggregated level-by-level at each node of the tree by summing the *demandIops* of its children and then applying the bound checks (equation 3) at the parent.

3.1.2 Computing Dynamic R,L,S for VMs

This step computes dynamic reservation, limit and share values for VMs based on the structure of the RP tree, the static (user-specified) reservation, limit and shares settings on the nodes, as well as the demand of VMs and internal nodes computed in Step 1. These operations are called R-*divvy*, L-*divvy* and S-*divvy* respectively. The exact divvy algorithm is explained in Section 3.2 followed by an example divvy computation in Section 3.3.

The R-divvy distributes the total reserved capacity at the root node among the currently active VMs. The allocation proceeds in a top-down hierarchical manner. First the root reservation is divided among its children based on their control settings. For the divvy, the limit of a child node is temporarily capped at its demand. This allows resources to preferentially flow to the nodes with higher current demand. Since the reservation at a node usually exceeds the sum of the reservations of its currently active children, the R-divvy will assign a higher reservation value per VM than its static setting.

The L-divvy is similarly used to provide higher dynamic limits to VMs with higher shares and demands. For instance, the user may set each VM limit to max (unlimited), but place an aggregate limit on the RP node. At run time, the aggregate limit needs to be allocated to individual VMs. The S-divvy similarly divides up the shares at a node among its children. Unlike the R and L divvies, the S-divvy does not use the demands but only the static share settings in doing the computation.

3.1.3 Update Array and Host Queue Depths

In this step, the SRP module computes the new array capacity, and the portion to be allocated to each host.

Since there is no centralized place to do scheduling across hosts, it is not possible to directly allocate IOPS to the hosts. Instead, we use the host queue depth (Q_h) as a control to do across-host allocation. We describe the three steps briefly below.

Update Array Queue Depth: To determine the new array queue depth we use a control strategy inspired by PARDA [6]. The queue depth is adjusted to keep the measured latency within the congestion threshold, using equation 4 below.

$$Q(t+1) = (1-\gamma)Q(t) \;+\; \gamma\left(\frac{\mathscr{L}_c}{Lat(t)}Q(t)\right) \quad (4)$$

Here $Q(t)$ denotes the array queue depth at time t, $Lat(t)$ is the current measured average latency, $\gamma \in [0,1]$ is a smoothing parameter, and $\mathscr{L}_c$ is the device congestion threshold.

Compute VM OIO Entitlement: We first convert the array queue depth value computed above to an equivalent array IOPS capacity using Little's law:

$$arrayIOPS = Q(t+1)/\mathscr{L}_c. \quad (5)$$

We then use the divvy algorithm (Algorithm 2) described in Section 3.2, to divide this capacity among all the VMs based on their settings. This results in the VM IOPS entitlement denoted by E_i. The conversion from queue depth to IOPS is done because the resource pool settings used for the divvy are in terms of user-friendly IOPS, rather than the less transparent OIO values.

Set Host Queue Depth: Finally, we set the host queue depth (Q_h) to be the fraction of the array queue depth that the host should get based on its share of the VM entitlements in the whole cluster (using equation 6).

$$Q_h = Q(t+1) \times \frac{\sum_{i \in VM\ on\ host} E_i}{arrayIOPS} \quad (6)$$

At each host, the local mClock scheduler is used to allocate the host's share of the array capacity (represented by the host queue depth Q_h) among its VMs. mClock uses the dynamic VM reservations, limits, and shares settings computed by SRP in step 2 to do the scheduling.

3.2 Divvy Algorithm

The root of the RP tree holds four resource types that need to be divided among the nodes of the tree: (1) RP reservation ($\mathscr{R}$), (2) RP limit ($\mathscr{L}$), (3) RP shares ($\mathscr{S}$), and (4) array IOPS. The first three values are divvied to compute the dynamic R,L,S settings, and the fourth value (array IOPS) is divvied to compute per VM entitlement. We use the same divvy algorithm for all these except for shares. The divvying of shares ($\mathscr{S}$) is much simpler and is based only on the static share values of the child nodes. The shares at a node are divided among the children in the ratio of the children's share settings. Next, we explain the common divvy algorithm for the remaining values. We use the generic term *capacity* to denote the resource being divvied.

Intuitively, the divvy will allocate the parent's capacity to its children in proportion to their shares, subject to their reservations and limit controls. Algorithm 2 presents an efficient algorithm to do the divvying for a given capacity $\mathscr{C}$. The goal is to assign each child to one of three sets: RB, LB, or PS. These represent children whose allocation either equals their reservation (RB), equals their limit (LB), or lies between the two (PS). The children in PS get allocations in proportion to their shares.

We use w_i to denote the fraction of shares assigned to child i relative to the total shares of all the children. We use the terms *normalized reservation* and *normalized limit* to denote the quantities r_i/w_i and l_i/w_i respectively. $\mathscr{V}$ is the ordered set of all normalized reservations and limits, arranged in increasing order. Ties between normalized reservation and limit values of child i are broken to ensure that r_i/w_i appears earlier than l_i/w_i.

Initially we allocate all children their reservations, and place them in set RB. At each step k, we see if there is enough capacity to increase the allocation of the current members of PS to a new target value v_k. This is either the normalized reservation or limit of some child denoted by $index[k]$. In the first case the child is moved from RB to PS, and in the latter case the child is moved from PS to LB. The total weight of the children in PS is adjusted accordingly. This continues till either the capacity is exhausted or all elements in $\mathscr{V}$ have been examined.

The complexity of the algorithm is $O(n \log n)$, bounded by the time to create the sorted sequence $\mathscr{V}$. At the end of the process, children in LB are allocated their limit, those in RB are allocated their reservation, and the rest receive allocation of the remaining capacity in proportion to their shares.

This divvy algorithm is used by the SRP module for R-divvy, L-divvy and entitlement computation. The only difference in these is the parameters with which the divvy algorithm is called. In all cases, the demand of a node is used as its temporary l value during the divvy, while its r and s values are the user set values. If the sum of the demands of the children is smaller than the capacity being divvied at the parent, the user set limits are used instead of the demand. For R-divvy, the reservation set at the root ($\mathscr{R}$) is used as the capacity to divvy, while for L-divvy and entitlement computation they are the root *limit* setting ($\mathscr{L}$) and the *arrayIOPS* respectively.

Algorithm 2: O(n log n) Divvy Algorithm

Data: $\mathscr{C}$: Capacity to divvy
Child c_i, $1 \le i \le n$, parameters: r_i, l_i, s_i.
Result: a_i: allocation computed for child c_i.
Variables: $w_i = s_i / \sum_{j=1}^{n} s_j$
$\mathscr{V}$: Ordered set $\{v_1, v_2, \cdots v_{2n},\ v_i \le v_{i+1}\}$ of elements from set $\{r_i/w_i,\ l_i/w_i,\ 1 \le i \le n\}$.
$index[i]$; equals k if v_i is either r_k or l_k.
$type[i]$: equals L (R) if v_i is a limit (reservation).
Sets: $RB = \{1, \cdots n\}$, $LB = \{\}$, $PS = \{\}$.
$RBcap = \sum_{j=1}^{n} r_j$, $LBcap = 0$, $PSwt = 0$.
foreach $k = 1, \cdots, 2n$ **do**
  /*Can allocation in PS be increased to v_k?*/
  if *($PSwt$ * v_k + $LBcap + RBcap > \mathscr{C}$)* **then**
    break
  /* If $type[k]$ is the limit of a child in PS: Transfer the child from PS set to LB set */
  if *($type[k] = L$)* **then**
    $LB = LB \cup \{index[k]\}$
    $LBcap = LBcap + l_{index[k]}$
    $PS = PS - \{index[k]\}$
    $PSwt = PSwt - w_{index[k]}$
  else
    /* $type[k]$ *is R*: Move child from RB to PS*/
    $PS = PS \cup \{index[k]\}$
    $PSwt = PSwt + w_{index[k]}$
    $RB = RB - \{index[k]\}$
    $RBcap = RBcap - r_{index[k]}$
if $i \in RB$ $a_i = r_i$; /*allocation equals reservation */
if $i \in LB$ $a_i = l_i$; /*allocation equals limit */
/* PS members get rest of capacity in shares ratio.*/
if $i \in PS$ $a_i = (w_i / \sum_{j \in PS} w_j) \times (\mathscr{C} - LB_{cap} - RB_{cap})$;

3.3 A Divvy Example

We illustrate the divvy operation using the RP tree in Figure 3. The tuple U denotes static *user* settings, and the tuple D shows the dynamic reservation, limit and share values of each node as computed by the *divvy* algorithm. The demand is also shown for each VM (leaf-nodes).
The first step is to aggregate VM demands (step 1 of Algorithm 1), and use it as a temporary cap on the limit settings of the nodes. Hence the limits on nodes A through D are temporarily set to 600, 400, 400, and 100 respectively. The limit for a non-leaf node is set to the smaller of its static limit and the sum of its children's limits. For nodes E and F the limits are set to 1000 and 500 respectively.

R Divvy: The algorithm then proceeds level-by-level down from the root using Algorithm 2 to divvy the parent reservation among its children. At the root of the tree, $\mathscr{R} = 1200$ is divvied between nodes E and F in the ratio of their shares 3 : 1, resulting in allocations of 900 and 300 respectively. Since these values lie between the reservation and limit values for the nodes, this is the final result of the R-divvy at the root node.

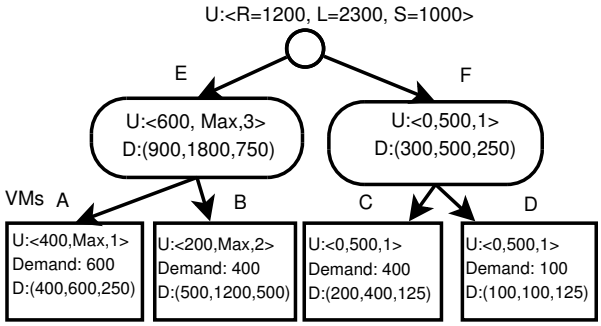


Figure 3: Divvy example for R, L and S

At the next level, the reservation of $R = 900$ at node E is divvied up among VMs A and B. Based on shares $(1 : 2)$, A would be allocated 300, which is below its reservation of 400. Hence, the algorithm would actually give A its reservation amount (400) and B would get the rest (500). For VMs C and D, divvying the parent reservation in the $1 : 1$ share ratio would lead to an allocation of 150 each; however, since D's limit has been temporarily capped at its demand, it is given 100 while C gets the remaining amount 200.
L Divvy: The L-divvy is similar and uses Algorithm 2 to divide the parent's limit among its children, level-by-level. The limit of $\mathscr{L}$ = 2300 at the root is divided in the ratio of $3 : 1$ among E and F, but is capped at the limit setting of the child; so the resulting allocations to nodes E and F are 1800 and 500 respectively. The dynamic limit settings at the other nodes can be similarly verified.
S-divvy: At each level the shares at the parent are simply divided in the ratio of the user set S values of the children.

Notice that although VMs C and D have identical static settings, due to the difference in their demands, the dynamic settings are different: $(200, 400, 125)$ and $(100, 100, 125)$ respectively. Similarly, excess reservation was given to VM B over VM A since it has a higher share value; however, to meet A's user-set reservation, B received less than twice A's reservation.

4 Implementation Issues

In this section, we discuss some of the implementation issues and storage-specific challenges that we handled while building our prototype for storage resource pools.

Shared Files. In order to share information across multiple hosts, we use three shared files on the underlying storage device running VMFS [1] clustered file system. The first file contains that structure of the resource pool tree and the static RP node settings. The second file allows hosts to share the current VM demands with each other. Each host is allotted a unique 512-byte block in

this file, that can be read by other hosts when performing the entitlement computation. Heart-beats using generation numbers are used in this file to detect host failures. The mapping of hosts to blocks is kept in a third file, and is the only structure that needs locking when a new host joins or leaves the resource pool.

This information could alternatively be disseminated via a broadcast or multicast network channel between the hosts. We chose to use shared files in our design because it reduces the dependence on other subsystems, so that a failure or congestion in the network does not affect SRP. Our approach only allocates 512 bytes per host and does 2 IOs per host every 4 seconds. We also use a special `MULTI_WRITER` mode for opening the file, which doesn't involve any locking. We have not seen any scalability issues in our testing of up to 32 hosts and don't expect scalability to be a major problem in the near future.

Local Scheduler. As shown in Figure 2, the mClock scheduler [8] is used on each host to allocate the host's resources among its VMs. In some cases, we noticed long convergence times before the scheduling reflected the new policy settings. To fix this, we modified the mClock algorithm to reset the internally used tags whenever control settings are changed by SRP. The update frequency of 4 seconds seems to work fine for mClock.

System Scalability. Scalability is a critical issue for VMware ESX server clusters that may support up to thousands of VMs. Two of the design choices that help us avoid potential performance bottlenecks are: (1) Every host makes its allocation decisions locally after reading the shared VM-demand file using a single IO. Having no central entity makes the system robust to host failures and, together with the efficient file access mentioned earlier, allows it to scale to a large number of hosts. (2) The implementation can also handle slightly stale VM data, and doesn't require a consistent snapshot of the per-VM demand values.

4.1 Storage-specific Challenges

A key question that arises in the SRP implementation is: *How many IOPS can we reserve on a storage device?* This $\mathscr{R}$ value for the root of the RP tree, is an upper bound on the total VM reservations that will be allowed on this LUN by admission control. This is a well-known (and difficult) problem since the throughput is highly dependent on the workload's access pattern.

We suggest and use the following approach in this work: compute the throughput (in IOPS) for the storage device using random read workloads. This can be done either at the time of installation or later by running a micro-benchmark. Some of the light-weight techniques proposed in Pesto [9] can also be used to determine this value. Once this is known, we use that as an upper bound on reservable capacity, providing a conservative bound for admission control and leaving some buffer capacity for use by VMs with no (or low) reservations.

IO sizes pose another problem in estimating the reservable capacity. To handle this we compute the value using a base IO size of 16KB, and treat large IO sizes as multiple IOs. Many functions can be used to determine this relationship as described in PARDA [6] and mClock [8]. We use a simple step function where we charge one extra IO for every 32 KB of IO size. This seems to provide a good approximation to more fine-grained approaches.

We also perform certain optimizations to help with sequential workloads. For instance, mClock schedules a batch of IOs from one VM if the IOs are close to each other (within 4 MB). Similarly, arrays try to detect sequential streams and do prefetching for them. In most virtual environments, however, blending of IOs happens at the array and sequentiality doesn't get preserved well at the backend. Features like thin-provisioning and deduplication also make it difficult to maintain sequentiality of IOs.

An interesting issue that needs to be faced when dealing with IO reservation is the concurrency required of the workload. If the array is being operated at a latency equal to the congestion threshold $\mathscr{L}_c$ and the reservation is R IOPS, steady-state operation requires the number of outstanding IOs to be $\mathscr{L}_c$ x R. For example, if the congestion threshold is 20ms, a single threaded VM application (doing synchronous IOs) can get a maximum throughput of 50 IOPS. In order to get a reservation of 200 IOPS, the application or VM should have 4 outstanding IOs most of the time. This issue is similar to meeting CPU reservations in a multi-vcpu VM. A VM with 8 1GHz virtual CPUs and 8 GHz reservation requires at least 8 active threads to get its full reservation.

Impact of SSD Storage. We expect SRP to work even better for SSD-based LUNs. First, the overall IOPS capacity is much higher and more predictable for SSDs, making it is easier to figure out the IOPS that can be reserved. Second, the issue of random vs. sequential IOs is also less pronounced in case of SSDs. Given the small response times, we can even run the algorithm more frequently than every 4 seconds to react faster to workload changes. This is something that we plan to explore as part of future work.

5 Experimental Evaluation

In this section, we present results from a detailed evaluation of our prototype implementation of storage resource pools in the VMware ESX server hypervisor [19]. Our experiments examine the the following four questions: (1) How well can SRP enforce resource controls (reservations, limits and shares) for VMs and resource pools

spanning multiple hosts? (2) How effective is SRP in flowing resources between VMs in the same pool? (3) How does SRP compare with PARDA and mClock running together? (4) How well do we handle enterprise workloads with dynamic behavior in terms of IO type, locality and IO sizes?

5.1 System Setup

We implemented storage resource pools as a user level process running on ESX, and implemented the necessary APIs to set mClock controls in the hypervisor. We use mClock as the underlying local IO scheduler in ESX. For the experiments, we used a cluster consisting of five ESX hosts accessing a shared storage array. Each host is a HP DL380 G5 with a dual-core Intel Xeon 3.0GHz processor, 16GB of RAM and two Qlogic HBAs connected to multiple arrays. We used two arrays for our experiments: (1) EMC CLARiiON CX over a FC SAN, and (2) Dell Equallogic array with iSCSI connection. The storage volumes are hosted on a 10-disk RAID-5 disk group on the EMC array, and a 15-disk (7 SSD, 8 SAS) pool on the Dell Equallogic array.

We used multiple micro and macro benchmarks running in separate VMs for our experiments. These include Iometer, DVDStore and Filebench based oltp, varmail and webserver workloads. The Iometer VMs have 1 virtual CPU, 1 GB RAM, 1 OS virtual disk of size 4GB and a 8 GB data disk. The DVDStore VM is a Windows Server 2008 machine with 2 virtual CPUs, 4 GB RAM, and three virtual disks of sizes 50 (OS disk), 25 (database disk) and 10 (log disk) GB respectively. The Filebench VMs have 4 virtual CPUs, 4 GB RAM and an OS disk of size 10 GB. For mail and webserver workloads we use a separate 16 GB virtual disk and for the oltp workload we use two separate disks of sizes 20 GB (data disk) and 1 GB (log disk) respectively.

5.2 Micro Benchmark Evaluations

In this section we present several experiments based on micro-benchmarks that show the effectiveness of SRP in doing allocations within and across resource pools.

5.2.1 Enforcement of Resource Pools Controls

First we show that the resource controls set at the VM and resource pool level are respected. For this experiment, we ran six VMs distributed across two hosts as shown in Figure 4. Host 1 runs VMs 1, 2 and 3, and the other VMs run in Host 2. All the VMs are accessing the EMC CLARiiON array.There are three resource pools RP1, RP2 and RP3 each with two VMs; all the resource pools have one VM on each of the two hosts. VMs 1 and 4 are in RP1, VMs 2 and 5 are in RP2 and VMs 3 and 6 are in RP3. The initial resource pool settings are shown in Table 2.

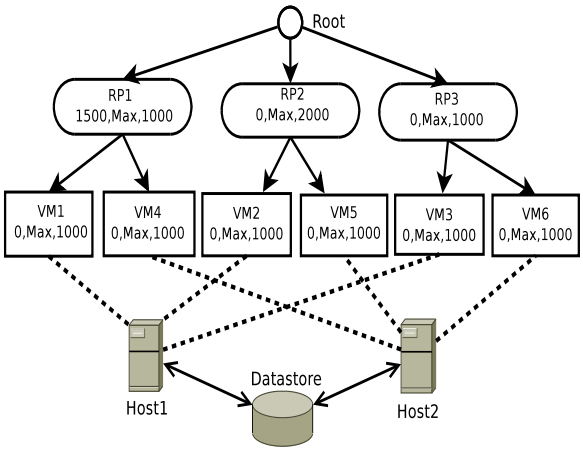


Figure 4: SRP tree configuration for micro benchmark based experiments

Resource Pool	Reservation	Shares	Limit	VMs
RP1	1500	1000	Max	1, 4
RP2	0	2000	Max	2, 5
RP3	0	1000	Max	3, 6

Table 2: Initial resource pool settings

VM	Size, Read%,Random%	Host	Resource Pool
VM1	4K, 75%, 100%	1	RP1
VM2	8K, 90%, 80%	1	RP2
VM3	16K, 75%, 20%	1	RP3
VM4	8K, 50%, 60%	2	RP1
VM5	16K, 100%, 100%	2	RP2
VM6	8K, 100%, 0%	2	RP3

Table 3: VM workload configurations

To demonstrate the practical effectiveness of SRP, we experimented with workloads having very different IO characteristics. We used six workloads, generated using Iometer on Windows VMs. All VMs are continuously backlogged with a fixed number of outstanding IOs. The workload configurations are shown in Table 3. The VMs do not have any reservation or limits set (which default to 0 and *max* respectively), and they all have equal shares of 1000.

At the start of the experiment, RP1 has a reservation of 1500 IOPS and 1000 shares. SRP should give RP1 its reservation of 1500 IOPS, and allocate additional capacity between RP2 and RP3 in the ratio of their shares (2 : 1), until their allocations catch up with RP1. The allocation to RP1 should be divided equally between VMs 1 and 4, which should receive allocation of 750 IOPS each.

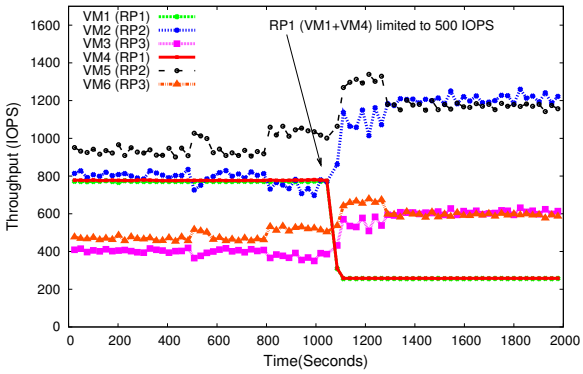


Figure 5: RP1's reservation is changed to 0 and its limit is set to 500 at t = 1000 sec. SRP always satisfies reservations and limits while doing allocation in proportion to shares.

Figure 5 shows the experimentally measured throughputs of all the VMs. The throughputs of VMs 1 and 4 are close to 750 IOPS as expected. The total throughput of RP2 (VMs 2 and 5) is a little less than twice that of RP3 (VMs 3 and 6). The reason is because VMs 3 and 6 have highly sequential workloads (80% and 100%), and get some preferential treatment from the array, resulting in a little higher throughput than their entitled allocations. After about 1000 seconds, the reservation of RP1 is set to 0 and its limit is reduced to 500 IOPS. Now VMs 1 and 4 only get 250 IOPS each, equally splitting the parent's limit of 500 IOPS. The rest of the capacity is divided between RP2 and RP3 as before in a rough 2 : 1 ratio.

We also experimented with setting the controls directly on the VMs instead of the RP nodes. We set reservations of 750 each for the VMs in RP1, and shares of 2000 (1000) to each of the VMs in RP2 (respectively RP3). The observed VM througphputs were similar to the initial portion of Figure 5. The ability to set controls at the RP nodes instead of individual VMs provides a very convenient way to share resources using very few explicit settings.

5.2.2 VM Isolation in Resource Pools

In this experiment we show how RPs allow for stronger sharing and better multiplexing among VMs in a pool so that resources stay within it. This has advantages when VMs are not continuously backlogged; the capacity freed up during idle periods is preferentially allocated to other (sibling) VMs within the pool rather than spread across all VMs.

The setup is identical to the previous experiment. At the start, the throughputs of the VMs shown in Figure 6 match the initial portion of Figure 5 as expected. Starting at time 250 seconds, VM 1 goes idle. We see that the entire slack is picked up by VM4, its sibling in RP1, whose throughput rises from 750 to 1500. The other VMs do not get to use this freed-up reservation since VM4 has first priority for it and it has enough demand to use it completely.

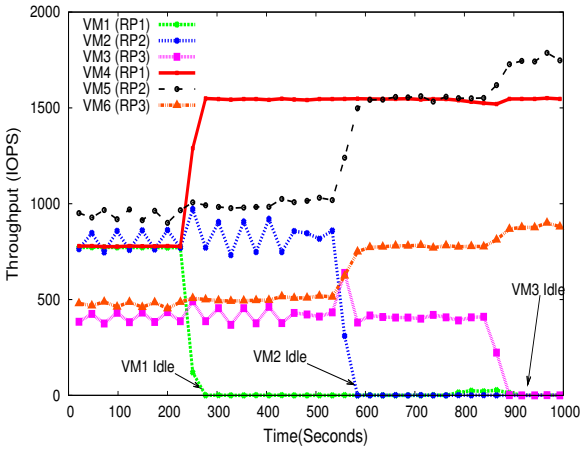


Figure 6: VMs 1, 2 and 3 get idle at t = 250, 500 and 750 sec respectively. Spare IOPS are allocated to the sibling VMs first

At time 550 seconds, VM2 goes idle, and its sibling VM5 on the other host sees the benefit within just a few seconds. VM6 which runs on the same host as VM5 also gets a slight boost from the increase in queue depth allocated to this host. The array also becomes more efficient and this benefit is given to all the active VMs in proportion to their shares. After VM2 becomes idle, RP2 gets higher IOPS than RP3 due to its higher shares.

Finally VM3 goes idle and VM6 gets the benefit. There is not much benefit to the other workloads when the sequential workload becomes idle. But still the reservations are always met and the workloads under RP2 and RP3 are roughly in the ratio of 2 : 1. This experiment shows the flow of resources within a resource pool and the isolation between pools.

5.2.3 Comparison with Parda+mClock

We compared Storage Resource Pool with a state-of-the-art system that supports reservation, limit and shares. We ran both PARDA [6] and mClock [8] together on ESX hosts. PARDA does proportional allocation for VMs based on the share settings. PARDA works across a cluster of hosts by leveraging a control algorithm that is very similar to FAST TCP [11] for flow control in networks. mClock is used as the local scheduler which supports reservations, limits and shares for VMs on the same host.

Since PARDA and mClock do not support resource pools, we created a single pool with VMs as its children and set the controls only at the VM-level. We found that

VM	R_i	L_i	S_i
VM1(Host1)	750	Max	1000
VM2(Host1)	750	Max	1000
VM3(Host1)	0	Max	1000
VM4(Host2)	0	Max	1000
VM5(Host2)	0	Max	1000
VM6(Host2)	0	Max	1000

Table 4: VM settings used for comparison of SRP versus PARDA+mClock

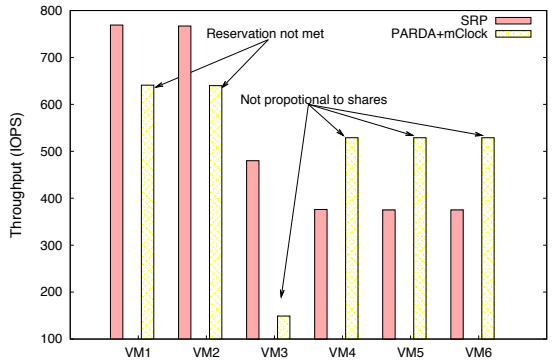


Figure 7: Comparison of throughput by SRP versus PARDA+mClock

even without the benefit of setting controls at the resource pool level, SRP is better than PARDA+mClock in two aspects. First PARDA+mClock could not satisfy VM reservations because PARDA adjusted the window sizes based solely on shares, while mClock tried to satisfy the reservation based on whatever window size was allocated by PARDA. This also showed that the local reservation penalized some VMs more than the others.

The second benefit of SRP is that when the settings are changed or when the workload changes in the VMs, SRP converges much faster than PARDA+mClock. We discuss each of these in more detail below.

Reservation Enforcement. We used six VMs running on two different ESX hosts. Per VM settings and VM-to-host mappings are shown in Table 4. We picked a simple case where a reservation of 750 IOPS was set for VMs 1 and 2. All the VMs have equal shares of 1000. VMs 1, 2 and 3 ran on host 1 and VMs 4, 5 and 6 ran on host 2. Both hosts ran PARDA and mClock initially. PARDA sees equal amount of shares on both hosts and allocates a host queue depth of 45 to both hosts. However, host 1 has two VMs with reservations of 750, and mClock tries to satisfy this by penalizing the third VM.

As seen in Figure 7, the third VM only gets 149 IOPS; the first two VMs are also not able to meet their reservations due to interference from host 2. When SRP is enabled in lieu of PARDA, it increases host 1's queue depth to 55 and reduces host 2's queue depth to 32. This enables VM1 and VM2 to meet their reservations and VM3 also gets IOPS that are much closer to VMs 4, 5 and 6. Thus with SRP the reservations are met and the other VMs get IOPS roughly in proportional to their shares.

Convergence Time. The response time to react to dynamic changes and converge to new settings is one of the critical performance factors in a distributed system. Quick convergence is usually desired to react to changes in a timely manner. In this section, we compare the convergence times of SRP to PARDA+mClock.

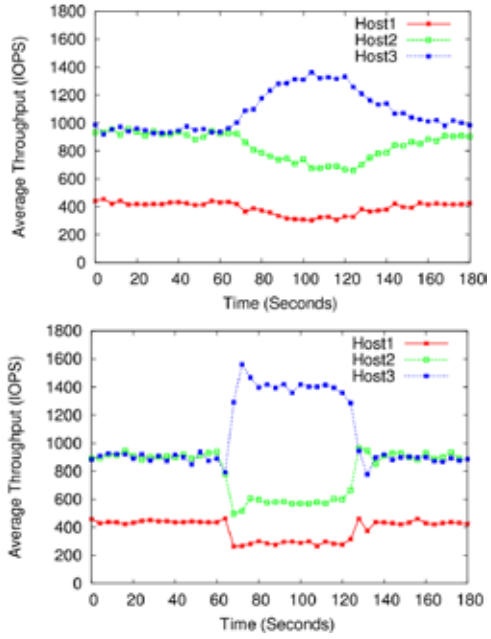


Figure 8: Average throughput of hosts while using (a) PARDA + mClock (top figure) and (b) SRP (bottom figure). SRP converges much faster as compared to PARDA+mClock.

To do the comparison, we ran VM1, VM2 and VM3 on separate hosts, with an initial shares ratio of 1 : 2 : 2. The VMs did not have reservations or limits set. Later the shares of VM3 were doubled at t = 60 second and reduced by half at t = 120 second. Each VM was running Windows IOmeter with the same configuration of 4KB, 100% random read, and 32 outstanding IOs at all times. Figure 8 shows the average throughput for the hosts using PARDA+mClock (top) and SRP (bottom) respectively. In general, both of the algorithms achieve resource sharing in proportion to their shares.

PARDA takes much longer (30 seconds) to converge to the new queue depth settings as compared to SRP (8 seconds). This is because PARDA runs a local control equation to react to global changes, where as SRP does a quick divvy of overall queue depth.

5.3 Enterprise Workloads

Next we tested storage resource pools for more realistic enterprise workloads with bursty arrivals, variable locality of reference, variable IO sizes and variable demand. We used four different workloads for this experiment: Filebench-Mail, Filebench-Webserver, Filebench-Oltp and DVDStore.

The first three workloads are based on different personalities of the well-known Filebench workload emulator [18], and the fourth workload is based on DVDStore [2] database test suite, which represents a complete online e-commerce application running on SQL database. For each VM, we used one OS disk and one or more data disks. These eight VMs, with a total of nineteen virtual disks, were spread across three ESX hosts accessing the same underlying device using VMFS.

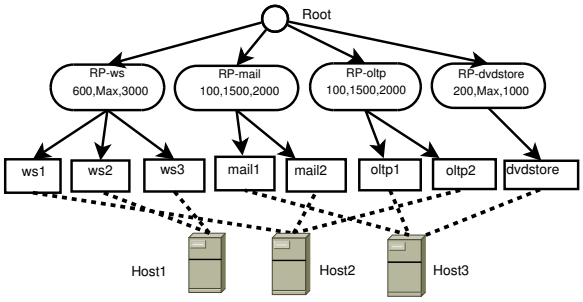


Figure 9: SRP configuration for Enterprise workloads

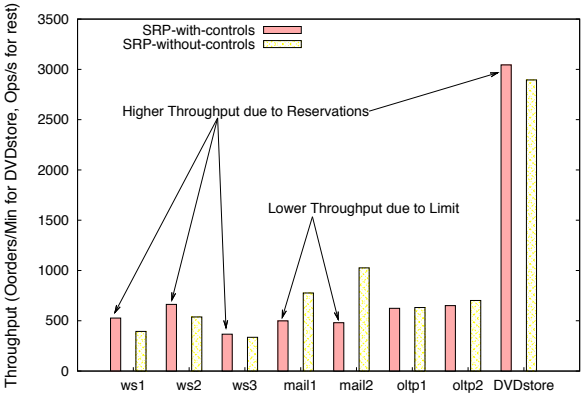


Figure 10: Comparison of application level throughput with and without SRP

For this experiment, we partitioned these four workloads into four different resource pools. These pools are called RP-mail, RP-oltp, RP-ws and RP-dvdstore respectively. RP-mail contains two VMs running the mail server workload, RP-ws contains three VMs running the webserver workload, RP-oltp contains two VMs running the oltp workload and RP-dvdstore contains one VM running the DVDStore workload.

Figure 9 shows the settings for these resource pools and the individual VMs. First we ran these workloads

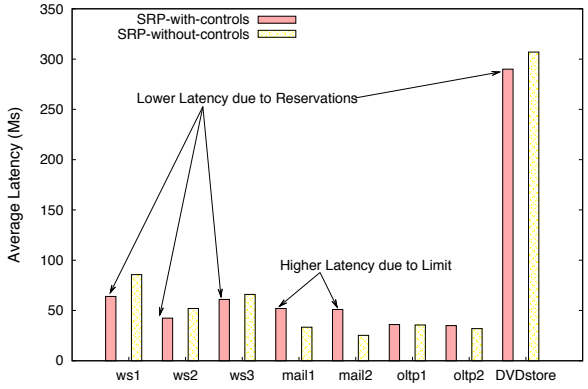


Figure 11: Comparison of application level average latency with and without SRP

with no reservation, infinite limit and equal shares. Then we set reservations and limit at the pool level to favor some workloads over others. The VM-level settings were unchanged. We set a reservation of 600 IOPS for webserver pool (RP-ws) and a reservation of 200 IOPS for RP-dvdstore. This reflects the user's concern that these workloads have a smaller latency. We also set a limit of 1500 IOPS for both RP-mail and RP-oltp pools. This is done to contain the impact of these very bursty VMs on others. We ran all these workloads together on the same underlying Equallogic datastore for 30 minutes, once for each setting.

Figures 10 and 11 show the overall application-level throughput in terms of Ops/sec (orders/min in case of DVDStore) and application-level average latency (in ms) for all VMs. Since we had set a reservation on the webserver and dvdstore pools, those VMs got lower latency and higher IOPS compared to other VMs. On the other hand the mail server VMs got higher latency because their aggregate demand is higher than the limit of 1500 IOPS. Interestingly, the effect on oltp VMs was much smaller because their overall demand is close to 1500 IOPS, so the limit didn't have as much of an impact.

This shows that by setting controls at the resource pool level, one can effectively isolate the workloads from one another. An advantage of setting controls at the resource pool level is that one doesn't have to worry about per VM controls, and VMs within a pool can take advantage of each other's idleness.

6 Conclusions

In this paper we studied the problem of doing hierarchical IO resource allocation in a distributed environment, where VMs running across multiple hosts are accessing the same underlying storage. We propose a novel and powerful abstraction called *storage resource pools* (SRP),

which allows setting of rich resource controls such as IO reservations, limits and proportional shares at VM or pool level. SRP does a two-level allocation by controlling per host queue depth and computing dynamic resource controls for VMs based on their workload demand using a resource divvying algorithm.

We implemented storage resource pools in VMware ESX hypervisor. Our evaluation with a diverse set of workloads shows that storage resource pools can guarantee high utilization of resources, while providing strong performance isolation for VMs in different resource pools. As future work, we plan to automate the resource control settings in order to provide application level SLOs and test our approach on multi-tiered storage devices.

Acknowledgments

We would like to thank Jyothir Ramanan, Irfan Ahmad, Murali Vilyannur, Chethan Kumar, and members of the DRS team, for discussions and help in setting up our test environment. We are also very grateful to our shepherd Rohit Chandra and anonymous reviewers for insightful comments and suggestions. We would also like to thank Bala Kaushik, Naveen Nagaraj, and Thuan Pham for discusions and motivating us to work on these problems.

References

[1] CLEMENTS, A. T., AHMAD, I., VILAYANNUR, M., AND LI, J. Decentralized Deduplication in SAN Cluster File Systems. In *Usenix ATC '09*.

[2] DELL. DVD Store. `http://www.delltechcenter.com/page/DVD+store`.

[3] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and simulation of a fair queuing algorithm. *Journal of Internetworking Research and Experience 1*, 1 (September 1990), 3–26.

[4] GOYAL, P., GUO, X., AND VIN, H. M. A hierarchical cpu scheduler for multimedia operating systems. In *Usenix OSDI* (January 1996).

[5] GOYAL, P., VIN, H. M., AND CHENG, H. Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks. Tech. Rep. CS-TR-96-02, UT Austin, January 1996.

[6] GULATI, A., AHMAD, I., AND WALDSPURGER, C. PARDA: Proportionate Allocation of Resources for Distributed Storage Access. In *Usenix FAST '09* (Feb. 2009).

[7] GULATI, A., MERCHANT, A., AND VARMAN, P. *p*Clock: An arrival curve based approach for QoS in shared storage systems. In *Proc. of ACM SIGMETRICS* (June 2007), pp. 13–24.

[8] GULATI, A., MERCHANT, A., AND VARMAN, P. J. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *Usenix OSDI'10* (Oct. 2010).

[9] GULATI, A., SHANMUGANATHAN, G., AHMAD, I., WALDSPURGER, C., AND UYSAL, M. PESTO: online Storage Performance Management in Virtualized Datacenters. In *Symposium on Cloud Computing (SOCC '11)* (Oct. 2011).

[10] HUANG, L., PENG, G., AND CKER CHIUEH, T. Multi-dimensional storage virtualization. In *ACM SIGMETRICS* (June 2004).

[11] JIN, C., WEI, D., AND LOW, S. FAST TCP: Motivation, Architecture, Algorithms, Performance. *Proceedings of IEEE INFOCOM* (March 2004).

[12] JIN, W., CHASE, J. S., AND KAUR, J. Interposed proportional sharing for a storage service utility. In *ACM SIGMETRICS* (June 2004), pp. 37–48.

[13] KARLSSON, M., KARAMANOLIS, C., AND ZHU, X. Triage: Performance differentiation for storage systems using adaptive control. *Trans. Storage 1*, 4 (2005), 457–480.

[14] LITTLE, J. D. C. A Proof for the Queuing Formula: $L = \lambda W$. *Operations Research 9*, 3 (1961).

[15] LUMB, C., MERCHANT, A., AND ALVAREZ, G. Façade: Virtual storage devices with performance guarantees. *Usenix FAST* (March 2003).

[16] POVZNER, A., KALDEWEY, T., BRANDT, S., GOLDING, R., WONG, T., AND MALTZAHN, C. Guaranteed Disk Request Scheduling with Fahrrad. In *EUROSYS* (March 2008).

[17] STOICA, I., ABDEL-WAHAB, H., AND JEFFAY, K. On the duality between resource reservation and proportional share resource allocation. *Multimedia Computing and Networking 3020* (1997), 207–214.

[18] SUN MICROSYSTEMS. Filebench Benchamarking Tool. `http://solarisinternals.com/si/tools/filebench/index.php`.

[19] VMWARE, INC. *Introduction to VMware Infrastructure*. 2007. http://www.vmware.com/support/pubs/.

[20] VMWARE, INC. *vSphere Resource Management Guide: ESX 4.1, ESXi 4.1, vCenter Server 4.1*. 2010.

[21] WACHS, M., ABD-EL-MALEK, M., THERESKA, E., AND GANGER, G. R. Argon: performance insulation for shared storage servers. In *FAST'07* (2007), pp. 5–5.

[22] WALDSPURGER, C. A. Memory Resource Management in VMware ESX Server. In *Usenix OSDI* (Dec. 2002).

[23] WANG, Y., AND MERCHANT, A. Proportional-share scheduling for distributed storage systems. In *Usenix FAST'07*.

[24] WONG, T. M., GOLDING, R. A., LIN, C., AND BECKER-SZENDY, R. A. Zygaria: Storage performance as managed resource. In *Proc. of RTAS* (April 2006), pp. 125–34.

[25] ZHANG, J., SIVASUBRAMANIAM, A., WANG, Q., RISKA, A., AND RIEDEL, E. Storage performance virtualization via throughput and latency control. In *Proc. of MASCOTS* (Sep 2005).

Erasure Coding in Windows Azure Storage

Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin
Microsoft Corporation

Abstract

Windows Azure Storage (WAS) is a cloud storage system that provides customers the ability to store seemingly limitless amounts of data for any duration of time. WAS customers have access to their data from anywhere, at any time, and only pay for what they use and store. To provide durability for that data and to keep the cost of storage low, WAS uses erasure coding.

In this paper we introduce a new set of codes for erasure coding called Local Reconstruction Codes (LRC). LRC reduces the number of erasure coding fragments that need to be read when reconstructing data fragments that are offline, while still keeping the storage overhead low. The important benefits of LRC are that it reduces the bandwidth and I/Os required for repair reads over prior codes, while still allowing a significant reduction in storage overhead. We describe how LRC is used in WAS to provide low overhead durable storage with consistently low read latencies.

1 Introduction

Windows Azure Storage (WAS) [1] is a scalable cloud storage system that has been in production since November 2008. It is used inside Microsoft for applications such as social networking search, serving video, music and game content, managing medical records, and more. In addition, there are thousands of customers outside Microsoft using WAS, and anyone can sign up over the Internet to use the system. WAS provides cloud storage in the form of Blobs (user files), Tables (structured storage), Queues (message delivery), and Drives (network mounted VHDs). These data abstractions provide the overall storage and work flow for applications running in the cloud.

WAS stores all of its data into an append-only distributed file system called the stream layer [1]. Data is appended to the end of active *extents*, which are replicated three times by the underlying stream layer. The data is originally written to 3 full copies to keep the data durable. Once reaching a certain size (e.g., 1 GB), extents are sealed. These sealed extents can no longer be modified and thus make perfect candidates for erasure coding. WAS then erasure codes a sealed extent lazily in the background, and once the extent is erasure-coded the original 3 full copies of the extent are deleted.

The motivation for using erasure coding in WAS comes from the need to reduce the cost of storage. Erasure coding can reduce the cost of storage over 50%, which is a tremendous cost saving as we will soon surpass an Exabyte of storage. There are the obvious cost savings from purchasing less hardware to store that much data, but there are significant savings from the fact that this also reduces our data center footprint by 1/2, the power savings from running 1/2 the hardware, along with other savings.

The trade-off for using erasure coding instead of keeping 3 full copies is performance. The performance hit comes when dealing with i) a lost or offline data fragment and ii) hot storage nodes. When an extent is erasure-coded, it is broken up into k data fragments, and a set of parity fragments. In WAS, a data fragment may be lost due to a disk, node or rack failure. In addition, cloud services are *perpetually in beta* [2] due to frequent upgrades. A data fragment may be offline for seconds to a few minutes due to an upgrade where the storage node process may be restarted or the OS for the storage node may be rebooted. During this time, if there is an on-demand read from a client to a fragment on the storage node being upgraded, WAS reads from enough fragments in order to dynamically reconstruct the data being asked for to return the data to the client. This reconstruction needs to be optimized to be as fast as possible and use as little networking bandwidth and I/Os as possible, with the goal to have the reconstruction time consistently low to meet customer SLAs.

When using erasure coding, the data fragment the client's request is asking for is stored on a specific storage node, which can greatly increase the risk of a storage node becoming hot, which could affect latency. One way that WAS can deal with hot storage nodes is to recognize the fragments that are hot and then replicate them to cooler storage nodes to balance out the load, or cache the data and serve it directly from DRAM or SSDs. But, the read performance can suffer for the potential set of reads going to that storage node as it gets hot, until the data is cached or load balanced. Therefore, one optimization WAS has is if it looks like the read to a data fragment is going to take too long, WAS in parallel tries to perform a reconstruction of the data fragment (effectively treating the storage node with the original data fragment as if it was offline) and return to the client whichever of the two results is faster.

For both of the above cases the time to reconstruct a data fragment for on-demand client requests is crucial. The problem is that the reconstruction operation is only as fast as the slowest storage node to respond to reading

of the data fragments. In addition, we want to reduce storage costs down to 1.33x of the original data using erasure coding. This could be accomplished using the standard approach of Reed-Solomon codes [13] where we would have (12, 4), which is 12 data fragments and 4 code fragments. This means that to do the reconstruction we would need to read from a set of 12 fragments. This i) greatly increases the chance of hitting a hot storage node, and ii) increases the network costs and I/Os and adds latency to read that many fragments to do the reconstruction. Therefore, we want to design a new family of codes to use for WAS that provides the following characteristics:

1. Reduce the minimal number of fragments that need to be read from to reconstruct a data fragment. This provides the following benefits: i) reduces the network overhead and number of I/Os to perform a reconstruction; ii) reduces the time it takes to perform the reconstruction since fewer fragments need to be read. We have found the time to perform the reconstruction is often dominated by the slowest fragments (the stragglers) to be read from.
2. Provide significant reduction in storage overhead to 1.33x while maintaining higher durability than a system that keeps 3 replicas for the data.

In this paper, we introduce Local Reconstruction Codes (LRC) that provide the above properties. In addition, we describe our erasure coding implementation and important design decisions.

2 Local Reconstruction Codes

In this section, we illustrate LRC and its properties through small examples, which are shorter codes (thus higher overhead) than what we use in production, in order to simplify the description of LRC .

2.1 Definition

We start with a Reed-Solomon code example to explain the concept of *reconstruction cost*. A (6, 3) Reed-Solomon code contains 6 data fragments and 3 parity fragments, where each parity is computed from all the 6 data fragments. When any data fragment becomes unavailable, no matter which data and parity fragments are used for reconstruction, 6 fragments are always required. We define *reconstruction cost* as the number of fragments required to reconstruct an unavailable *data fragment*. Here, the reconstruction cost equals to 6.

The goal of LRC is to reduce the reconstruction cost. It achieves this by computing some of the parities from a subset of the data fragments. Continuing the example with 6 data fragments, LRC generates 4 (instead of 3) parities. The first two parities (denoted as p_0 and p_1) are *global parities* and are computed from *all* the data fragments. But, for the other two parities, LRC divides the

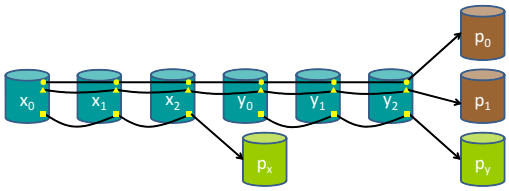


Figure 1: **A (6, 2, 2) LRC Example.** ($k = 6$ data fragments, $l = 2$ local parities and $r = 2$ global parities.)

data fragments into two equal size groups and computes one *local parity* for each group. For convenience, we name the 6 data fragments (x_0, x_1 and x_2) and (y_0, y_1 and y_2). Then, local parity p_x is computed from the 3 data fragments in one group (x_0, x_1 and x_2), and local parity p_y from the 3 data fragments in another group (y_0, y_1 and y_2).

Now, let's walk through reconstructing x_0. Instead of reading p_0 (or p_1) and the other 5 data fragments (x_1, x_2, y_0, y_1 and y_2), it is more efficient to read p_x and two data fragments (x_1 and x_2) to compute x_0. It is easy to verify that the reconstruction of *any* single data fragment requires only 3 fragments, half the number required by the Reed-Solomon code.

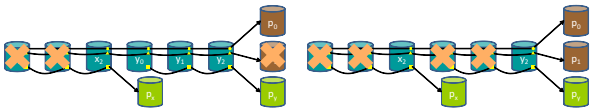
This LRC example adds one more parity than the Reed-Solomon one, so it might appear that LRC reduces reconstruction cost at the expense of higher storage overhead. In practice, however, these two examples achieve completely different trade-off points in the design space, as described in Section 3.3. In addition, LRC provides more options than Reed-Solomon code, in terms of trading off storage overhead and reconstruction cost.

We now formally define Local Reconstruction Codes. A (k, l, r) LRC divides k data fragments into l groups, with k/l data fragments in each group. It computes one local parity within each group. In addition, it computes r global parities from all the data fragments. Let n be the total number of fragments (data + parity). Then $n = k + l + r$. Hence, the normalized storage overhead is $n/k = 1 + (l + r)/k$. The LRC in our example is a $(6, 2, 2)$ LRC with storage cost of $1 + 4/6 = 1.67$x, as illustrated in Figure 1.

2.2 Fault Tolerance

Thus far, we have only defined which data fragments are used to compute each parity in LRC. To complete the code definition, we also need to determine *coding equations*, that is, how the parities are computed from the data fragments. We choose the coding equations such that LRC can achieve the *Maximally Recoverable* (MR) property [14], which means it can decode any failure pattern which is information-theoretically decodable.

Let's first explain the Maximally Recoverable property. Given the (6, 2, 2) LRC example, it contains 4 parity fragments and can tolerate *up to* 4 failures. However,



(a) 3 Failures (b) 4 Failures

Figure 2: Decoding 3 and 4 Failures in LRC.

LRC is not Maximum Distance Separable [12] and therefore cannot tolerate *arbitrary* 4 failures. For instance, say the 4 failures are x_1, x_2, x_3 and p_x. This failure pattern is non-decodable because there are only two parities - the global parities - that can help to decode the 3 missing data fragments. The other local parity p_y is useless in this example. It is *impossible* to decode 3 data fragments from merely 2 parity fragments, regardless of coding equations. These types of failure patterns are called *information-theoretically* non-decodable.

Failure patterns that are possible to reconstruct are called *information-theoretically* decodable. For instance, the 3-failure pattern in Figure 2(a) and the 4-failure pattern in Figure 2(b) are both information-theoretically decodable. For these two failure patterns, it is possible to construct coding equations such that it is equivalent to solving 3 unknowns using 3 linearly independent equations in Figure 2(a) and 4 unknowns using 4 linearly independent equations in Figure 2(b).

Conceivably, it is not difficult to construct a set of coding equations that can decode a specific failure pattern. However, the real challenge is to construct a *single* set of coding equations that achieves the *Maximally Recoverable* (MR) property [14], or being able to decode all the information-theoretically decodable failure patterns – the exact goal of LRC.

2.2.1 Constructing Coding Equations

It turns out that the LRC can tolerate arbitrary 3 failures by choosing the following two sets of coding coefficients (α's and β's) for group x and group y, respectively. We skip the proofs due to space limitation. Let

$$q_{x,0} = \alpha_0 x_0 + \alpha_1 x_1 + \alpha_2 x_2 \quad (1)$$

$$q_{x,1} = \alpha_0^2 x_0 + \alpha_1^2 x_1 + \alpha_2^2 x_2 \quad (2)$$

$$q_{x,2} = x_0 + x_1 + x_2 \quad (3)$$

and

$$q_{y,0} = \beta_0 y_0 + \beta_1 y_1 + \beta_2 y_2 \quad (4)$$

$$q_{y,1} = \beta_0^2 y_0 + \beta_1^2 y_1 + \beta_2^2 y_2 \quad (5)$$

$$q_{y,2} = y_0 + y_1 + y_2. \quad (6)$$

Then, the LRC coding equations are as follows:

$$p_0 = q_{x,0} + q_{y,0}, \quad p_1 = q_{x,1} + q_{y,1}, \quad (7)$$

$$p_x = q_{x,2}, \quad p_y = q_{y,2}. \quad (8)$$

Next, we determine the values of α's and β's so that the LRC can decode all information-theoretically decodable 4 failures. We focus on non-trivial cases as follows:

1. **None of the four parities fails.** The four failures are equally divided between group x and group y. Hence, we have four equations whose coefficients are given by the matrix, which result in the following determinant:

$$G = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ \alpha_i & \alpha_j & \beta_s & \beta_t \\ \alpha_i^2 & \alpha_j^2 & \beta_s^2 & \beta_t^2 \end{pmatrix}$$

$$Det(G) = (\alpha_j - \alpha_i)(\beta_t - \beta_s)(\alpha_i + \alpha_j - \beta_s - \beta_t).$$

2. **Only one of p_x and p_y fails.** Assume p_y fails. For the remaining three failures, two are in group x and the third one in group y. We now have three equations with coefficients given by

$$G' = \begin{pmatrix} 1 & 1 & 0 \\ \alpha_i & \alpha_j & \beta_s \\ \alpha_i^2 & \alpha_j^2 & \beta_s^2 \end{pmatrix}$$

$$Det(G') = \beta_s(\alpha_j - \alpha_i)(\beta_s - \alpha_j - \alpha_i).$$

3. **Both p_x and p_y fail.** In addition, the remaining two failures are divided between group x and group y. We have two equations with coefficients given by

$$G'' = \begin{pmatrix} \alpha_i & \beta_s \\ \alpha_i^2 & \beta_s^2 \end{pmatrix}$$

$$Det(G'') = \alpha_i \beta_s (\beta_s - \alpha_i).$$

To ensure all the cases are decodable, all the matrices G, G' and G'' should be non-singular, which leads to the following conditions:

$$\alpha_i,\ \alpha_j,\ \beta_s,\ \beta_t \neq 0 \quad (9)$$

$$\alpha_i,\ \alpha_j \neq \beta_s,\ \beta_t \quad (10)$$

$$\alpha_i + \alpha_j \neq \beta_s + \beta_t \quad (11)$$

One way to fulfill these conditions is to assign to α's and β's the elements from a finite field GF(2^4) [12], where every element in the field is represented by 4 bits. α's are chosen among the elements whose lower order 2 bits are zero. Similarly, β's are chosen among the elements whose higher order 2 bits are zero. That way, the lower order 2 bits of α's (and the sum of α's) are always zero, and the higher order 2 bits of β's (and the sum of β's) are always zero. Hence, they will never be equal and all the above conditions are satisfied.

This way of constructing coding equations requires a very small finite field and makes implementation practical. It is a critical improvement over our own Pyramid

codes [14]. In Pyramid codes, coding equation coefficients are discovered through a search algorithm, whose complexity grows exponentially with the length of the code. For parameters considered in Windows Azure Storage, the search algorithm approach would have resulted in a large finite field, which makes encoding and decoding complexity high.

2.2.2 Putting Things Together

To summarize, the $(6, 2, 2)$ LRC is capable of decoding arbitrary three failures. It can also decode all the information-theoretically decodable four failure patterns, which accounts for 86% of all the four failures. In short, the $(6, 2, 2)$ LRC achieves the Maximally Recoverable property [14].

2.2.3 Checking Decodability

Given a failure pattern, how can we easily check whether it is information-theoretically decodable? Here is an efficient algorithm. For each local group, if the local parity is available, while at least one data fragment is erased, we *swap* the parity with one erased data fragment. The swap operation marks the data fragment as available and the parity as erased. Once we complete all the local groups, we examine the data fragments and the global parities. If the total number of erased fragments (data and parity) is no more than the number of global parities, the algorithm declares the failure pattern information-theoretically decodable. Otherwise, it is non-decodable. This algorithm can be used to verify that the examples in Figure 2 are indeed decodable.

2.3 Optimizing Storage Cost, Reliability and Performance

Throughout the entire section, we have been focusing on the (6, 2, 2) LRC example. It is important to note that all the properties demonstrated by the example generalize to arbitrary coding parameters.

In general, the key properties of a (k, l, r) LRC are: i) single data fragment failure can be decoded from k/l fragments; ii) arbitrary failures up to $r + 1$ can be decoded. Based on the following theorem, these properties impose a lower bound on the number of parities [21].

Theorem 1. *For any (n, k) linear code (with k data symbols and $n - k$ parity symbols) to have the property:*

1. *arbitrary $r + 1$ symbol failures can be decoded;*
2. *single data symbol failure can be recovered from $\lceil k/l \rceil$ symbols,*

the following condition is necessary:

$$n - k \geq l + r. \tag{12}$$

Since the number of parities of LRC meets the lower bound *exactly*, LRC achieves its properties with the minimal number of parities.

2.4 Summary

Now, we summarize Local Reconstruction Codes. A (k, l, r) LRC divides k data fragments into l local groups. It encodes l local parities, one for each local group, and r global parities. Any single data fragment failure can be decoded from k/l fragments within its local group.

In addition, LRC achieves Maximally Recoverable property. It tolerates up to $r + 1$ arbitrary fragment failures. It also tolerates failures more than $r+1$ (up to $l+r$), provided those are information-theoretically decodable.

Finally, LRC provides low storage overhead. Among all the codes that can decode single data fragment failure from k/l fragments and tolerate $r + 1$ failures, LRC requires the minimum number of parities.

3 Reliability Model and Code Selection

There are many choices of parameters k, l and r for LRC. The question is: what parameters should we choose in practice? To answer this, we first need to understand the reliability achieved by each set of parameters. Since 3-replication is an accepted industry standard, we use the reliability of 3-replication as a reference. Only those sets of parameters that achieve equal or higher reliability than 3-replication are considered.

3.1 Reliability Model

Reliability has long been a key focus in distributed storage systems [28, 29, 30, 31]. Markov models are commonly used to capture the reliability of distributed storage systems. The model is flexible to consider both independent or correlated failures [10, 32, 33, 34]. We add a simple extension to generalize the Markov model, in order to capture unique state transitions in LRC. Those transitions are introduced because the failure mode depends on not only the size of failure, but also which subset of nodes fails. In our study, we focus on independent failures, but the study can be readily generalized to correlated failures [10, 34].

3.1.1 Modeling (6, 2, 2) LRC

We start with the standard Markov model to analyze reliability. Each state in the Markov process represents the number of available fragments (data and parity). For example, Figure 3 plots the Markov model diagram for the $(6, 2, 2)$ LRC.

Let λ denote the failure rate of a single fragment. Then, the transition rate from all fragments healthy `State 10` to one fragment failure `State 9` is 10λ. The extension from the standard Markov model lies in `State 7`, which can transition into two states with 6 healthy fragments. `State 6` represents a state where there are four decodable failures. On the other hand, `State 6F` represents a state with four non-decodable failures. Let p_d denote the percentage of decodable four failure cases. Then the transition rate from `State 7` to

`State 6` is $7\lambda p_d$ and the transition to `State 6F` is $7\lambda(1-p_d)$.

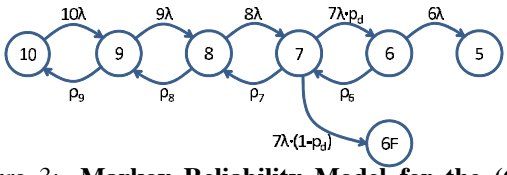


Figure 3: **Markov Reliability Model for the (6, 2, 2) LRC.** (State represents the number of healthy fragments.)

In the reverse direction, let ρ_9 denote the transition rate from one fragment failure `state 9` back to all fragments healthy `state 10`, which equals to the average repair rate of single-fragment failures.

Assume there are M storage nodes in the system, each with S storage space and B network bandwidth. When single storage node fails, assume the repair load is evenly distributed among the remaining $(M-1)$ nodes. Further, assume repair traffic is throttled so that it only uses ϵ of the network bandwidth on each machine. When erasure coding is applied, repairing one failure fragment requires more than one fragment, denoted as repair cost C. Then, the average repair rate of single-fragment failures is $\rho_9 = \epsilon(M-1)B/(SC)$.

Other transition rates ρ_8 through ρ_6 can be calculated similarly. In practice, however, the repair rates beyond single failure are dominated by the time taken to detect failure and trigger repair.[1] Let T denote the detection and triggering time. It suffices to set $\rho_8 = \rho_7 = \rho_6 = 1/T$.

Any single storage node stores both data and parity fragments from different extents. So, when it fails, both data and parity fragments need to be repaired. The repair cost can be calculated by averaging across all the fragments.

For the $(6, 2, 2)$ LRC, it takes 3 fragments to repair any of the 6 data fragments and the 2 local parities. Further, it takes 6 fragments to repair the 2 global parities. Hence, the average repair cost $C = (3 \times 8 + 6 \times 2)/10 = 3.6$. Also, enumerating all four failure patterns, we obtain the decodability ratio as $p_d = 86\%$.

3.1.2 Reliability of (6, 2, 2) LRC

Now, we use a set of typical parameters ($M = 400$, $S = 16$TB, $B = 1$Gbps, $\epsilon = 0.1$ and $T = 30$ minutes) to calculate the reliability of the LRC, which is also compared to 3-replication and Reed-Solomon code in Table 1. Since the Reed-Solomon code tolerates three failures, while 3-replication tolerates only two failures, it should be no surprise that the Reed-Solomon code offers higher reliability than 3-replication. Similarly, the LRC tolerates not only three failures, but also 86% of the four-failure cases, so it naturally achieves the highest reliability.

[1] When multiple failures happen, most affected coding groups only have a single fragment loss. *Unlucky* coding groups with two or more fragment losses are relatively few. Therefore, not many fragments enter multi-failure repair stages. In addition, multi-failure repairs are prioritized over single-failure ones. As a result, multi-failure repairs are fast and they take very little time, compared to detecting the failures and triggering the repairs.

	MTTF (years)
3-replication	3.5×10^9
(6, 3) Reed-Solomon	6.1×10^{11}
(6, 2, 2) LRC	2.6×10^{12}

Table 1: Reliability of 3-Replication, RS and LRC.

3.2 Cost and Performance Trade-offs

Each set of LRC parameters (k, l and r) yields one set of values of reliability, reconstruction cost and storage overhead. For the $(6, 2, 2)$ LRC, the reliability (MTTF in years) is 2.6×10^{12}, the reconstruction cost is 3 and the storage overhead is 1.67x.

We obtain many sets of values by varying the parameters. Since each fragment has to place on a different fault domain, the number of fault domains in a cluster limits the total number of fragments in the code. We use 20 as the limit here, since our storage stamps (clusters) have up to 20 fault domains. Using the reliability (MTTF) of 3-replication as the threshold, we keep those sets of parameters that yield equal or higher reliability than 3-replication. We then plot the storage overhead and the reconstruction cost of the remaining sets in Figure 4. Again, each individual point represents one set of coding parameters. Each parameter set represents certain trade-offs between storage cost and reconstruction performance.

Different coding parameters can result in the same storage overhead (such as 1.5x), but vastly different reconstruction cost. In practice, it only makes sense to choose the one with the lower reconstruction cost. Therefore, we outline the lower bound of all the trade-off points. The lower bound curve characterizes the fundamental trade-off between storage cost and reconstruction performance for LRC.

3.3 Code Parameter Selection

Similarly, for (k, r) Reed-Solomon code, we vary the parameters k and r (so long as $k + r \leq 20$) and also obtain a lower bound cost and performance trade-off curve. We compare Reed-Solomon to LRC in Figure 5. On the Reed-Solomon curve, we mark two special points, which are specific parameters chosen by existing planet-scale cloud storage systems. In particular, RS (10, 4) is used in HDFS-RAID in Facebook [8] and RS (6, 3) in GFS II in Google [9, 10]. Again, note that all the trade-off points achieve higher reliability than 3-replication, so they are

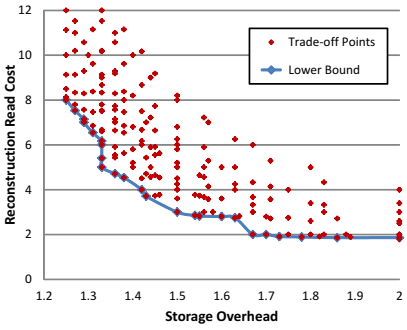


Figure 4: Overhead vs. Recon. Cost.

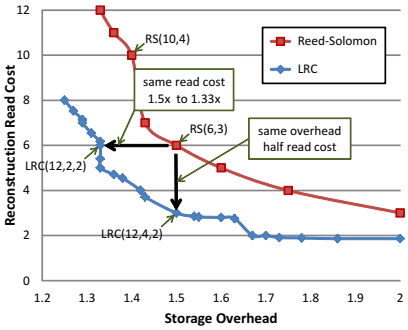


Figure 5: LRC vs. RS Code.

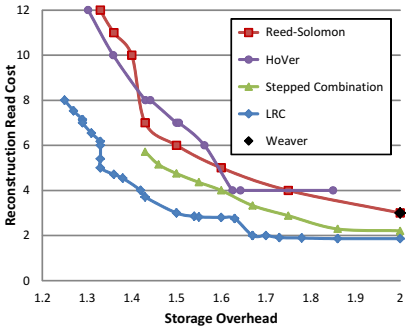


Figure 6: LRC vs. Modern Codes.

candidates for WAS.

The comparison shows that LRC achieves better cost and performance trade-off than Reed-Solomon across the range of parameters. If we keep the storage overhead the same, reconstruction in LRC is much more efficient than that in Reed-Solomon. On the other hand, if we keep reconstruction cost the same, LRC can greatly reduce storage overhead, compared to Reed-Solomon. In general, we can choose a point along the cost and performance trade-off curve, which can reduce storage overhead and reconstruction cost at the same time.

Compared to (6, 3) Reed-Solomon, we could keep storage overhead the same (at 1.5x) and replace the Reed-Solomon code with a (12, 4, 2) LRC. Now, the reconstruction cost is reduced from 6 to 3 for single-fragment failures, a reduction of 50%. This is shown by the vertical move in Figure 5.

Alternatively, as shown by the horizontal move in Figure 5, we could keep reconstruction cost the same (at 6) and replace the Reed-Solomon code with a (12, 2, 2) LRC. Now, the storage overhead is reduced from 1.5x to 1.33x. For the scale of WAS, such reduction translates into significant savings.

3.4 Comparison - Modern Storage Codes

We apply the same reliability analysis to state-of-the-art erasure codes designed specifically for storage systems, such as Weaver codes [18], HoVer codes [19] and Stepped Combination codes [20]. We examine the trade-off between reconstruction cost and storage overhead when these codes are at least as reliable as 3-replication.

The results are plotted in Figure 6. The storage overhead of Weaver codes is 2x or higher, so only a single point (storage overhead = 2x, reconstruction cost = 3) is shown in the Figure. We observe that the trade-off curve of Stepped Combination codes is strictly below that of Reed-Solomon. Therefore, both codes can achieve more efficient reconstruction than Reed-Solomon when storage overhead is fixed. Similarly, they require less storage overhead when reconstruction cost is fixed. HoVer codes offer many trade-off points better than Reed-Solomon.

Compared to these modern storage codes, LRC is superior in terms of the trade-off between reconstruction cost and storage overhead. The primary reason is that LRC separates parity fragments into local ones and global ones. In this way, local parities only involve minimum data fragments and can thus be most efficient for providing reconstruction reads when there is a hot spot, for reconstructing single failures, and for reconstructing a single fragment that is offline due to upgrade. On the other hand, global parities involve all the data fragments and can thus be most useful to provide fault tolerance when there are multiple failures. In contrast, in Weaver codes, HoVer codes and Stepped Combination codes, all parities carry both the duty of reconstruction and fault tolerance. Therefore, they cannot achieve the same trade-off as LRC.

LRC is optimized for reconstructing data fragments, but not parities, in order to quickly reconstruct on-demand based reads from clients. In terms of parity reconstruction, Weaver codes, HoVer codes and Stepped Combination codes can be more efficient. For instance, let's compare Stepped Combination code to LRC at the storage overhead of 1.5x, where both codes consist of 12 data fragments and 6 parities. For the Stepped Combination code, every parity can be reconstructed from 3 fragments, while for LRC the reconstruction of global parities requires as many as 12 fragments. It turns out that there is fundamental contention between parity reconstruction and data fragment reconstruction, which we studied in detail separately [21]. In WAS, since parity reconstruction happens only when a parity is lost (e.g., disk, node or rack failure), it is off the critical path of serving client reads. Therefore, it is desirable to trade the efficiency of parity reconstruction in order to improve the performance of data fragment reconstruction, as is done in LRC.

3.5 Correlated Failures

The Markov reliability model described in Section 3.1 assumes failures are independent. In practice, correlated failures do happen [10, 34]. One common correlated fail-

ure source is all of the servers under the same fault domain. WAS avoids these correlated failures by always placing fragments belonging to the same coding group in different fault domains.

To account for additional correlated failures, the Markov reliability model can be readily extended by adding transition arcs between non-adjacent states [10].

4 Erasure Coding Implementation in WAS

The WAS architecture has three layers within a storage stamp (cluster) - front-end layer, partitioned object layer and stream replication layer [1].

Erasure coding in WAS is implemented in the stream layer as a complementary technique to full data replication. It is also possible to implement erasure coding across multiple storage stamps on several data centers [38]. Our choice to implement erasure coding inside the stream layer is based on the fact that it fits the overall WAS architecture where the stream layer is responsible for keeping the data durable within a stamp and the partition layer is responsible for geo-replicating the data between data centers (see [1] for more details).

4.1 Stream Layer Architecture

The main components of the stream layer are the Stream Managers (SM), which is a Paxos [37] replicated server, and Extent Nodes (EN) (see Figure 7).

Streams used by the partition layer are saved as a list of extents in the stream layer. Each extent consists of a list of append blocks. Each block is CRC'd and this block is the level of granularity the partitioned object layer uses for appending data to the stream, as well as reading data (the whole block is read to get any bytes out of the block, since the CRC is checked on every read). Each extent is replicated on multiple (usually three) ENs. Each write operation is committed to all nodes in a replica set in a daisy chain, before an acknowledgment is sent back to the client. Write operations for a stream keep appending to an extent until the extent reaches its maximum size (in the range of 1GB-3GB) or until there is a failure in the replica set. In either case, a new extent on a new replica set is created and the previous extent is sealed. When an extent becomes sealed, its data is immutable, and it becomes a candidate for erasure coding.

4.2 Erasure Coding in the Stream Layer

The erasure coding process is completely asynchronous and off the critical path of client writes. The SM periodically scans all sealed extents and schedules a subset of them for erasure coding based on stream policies and system load. We configure the system to automatically erasure code extents storing Blob data, but also have the option to erasure code Table extents too.

As a first step of erasure coding of an extent, the SM creates fragments on a set of ENs whose number depends

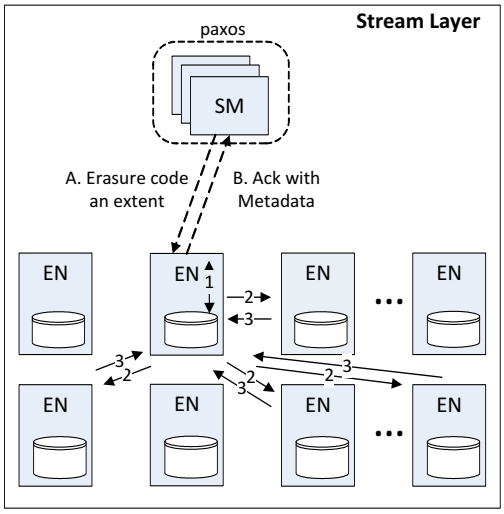


Figure 7: **Erasure Coding of an Extent** (not all target ENs are shown).

on the erasure coding parameters, for example 16 fragments for LRC (12, 2, 2).

The SM designates one of the ENs in the extent's replica set as the coordinator of erasure coding (see Figure 7) and sends it the metadata for the replica set. From then on, the coordinator EN has the responsibility of completing the erasure coding. The coordinator EN has, locally on its node, the full extent that is to be erasure-coded. The EN prepares the extent for erasure coding by deciding where the boundaries for all of the fragments will be in the extent. It chooses to break the extent into fragments at append block boundaries and not at arbitrary offsets. This ensures that reading a block will not cross multiple fragments.

After the coordinator EN decides what the fragment offsets are, it communicates those to the target ENs that will hold each of the data and parity fragments. Then the coordinator EN starts the encoding process and keeps sending the encoded fragments to their designated ENs. The coordinator EN, as well as each target EN, keeps track of the progress made and persists that information into each new fragment. If a failure occurs at any moment in this process, the rest of the work can be picked up by another EN based on the progress information persisted in each fragment. After an entire extent is coded, the coordinator EN notifies the SM, which updates the metadata of the extent with fragment boundaries and completion flags. Finally, the SM schedules full replicas of the extent for deletion as they are no longer needed.

The fragments of an extent can be read directly by a client (i.e., the partition or front-end layer in WAS) by contacting the EN that has the fragment. However, if that target EN is not available or is a hot spot, the client can contact any of the ENs that has any of the fragments of the extent, and perform a reconstruction read (see Fig-

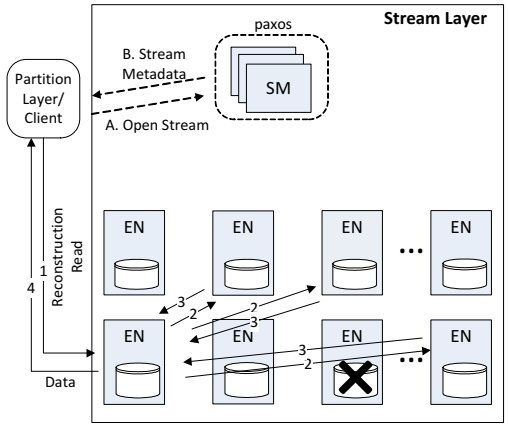


Figure 8: **Reconstruction for On-Demand Read** (not all target ENs are shown).

ure 8). That EN would read the other needed fragments from other ENs, then reconstruct that fragment, cache the reconstructed fragment locally in case there are other reads to it, and return the results to the client. LRC reduces the cost of this reconstruction operation considerably by reducing the number of source ENs that need to be accessed.

If the EN or the disk drive that hosts the extent fragment is unavailable for an extended period of time, the SM initiates the reconstruction of the fragment on a different EN. This operation is almost identical to the reconstruction steps shown in Figure 8 except for the fact that the operation is initiated by the SM instead of the client, and the data is written to disk rather than being sent back to the client.

4.3 Using Local Reconstruction Codes in Windows Azure Storage

When LRC is used as an erasure coding algorithm, each extent is divided into k equal-size *data* fragments. Then l local and r global *parity* fragments are created.

The placement of the fragments takes into account two factors: i) load, which favors less occupied and less loaded extent nodes; ii) reliability, which avoids placing two fragments (belonging to the same erasure coding group) into the same correlated domain. There are two primary correlated domains: fault domain and upgrade domain. A fault domain, such as rack, categorizes a group of nodes which can fail together due to common hardware failure. An upgrade domain categorizes a group of nodes which are taken offline and upgraded at the same time during each upgrade cycle. Upgrade domains are typically orthogonal to fault domains.

Let's now use LRC (12, 2, 2) as an example and illustrate an actual fragment placement in WAS. A WAS stamp consists of 20 racks. For maximum reliability, each of the total 16 fragments for an extent is placed in a different rack. If each fragment were similarly placed on a different upgrade domain, then at least 16 upgrade domains are required. In practice, however, too many upgrade domains can slow down upgrades and it is desirable to keep the number of upgrade domains low. In WAS, we currently use 10 upgrade domains. This means that we have at most 10% of the storage stamps (cluster) resources offline at any point in time during a rolling upgrade.

Given our desire to have fewer upgrade domains than fragments, we need an approach for placing the fragments across the upgrade domains to still allow LRC to perform its fast reconstruction for the fragments that are offline. To that end, we exploit the local group property of LRC and group fragments belonging to different local groups into same upgrade domains. In particular, we place the two local groups x and y so that their data fragments x_i and y_i are in the same upgrade domain i (i.e., x_0 and y_0 are placed in the same upgrade domain, but different fault domains). Similarly, we place the local parities p_x and p_y in one upgrade domain as well. The two global parities are placed in two separate upgrade domains from all other fragments.

Take for example LRC (12, 2, 2). In total, we use 9 upgrade domains for placing the fragments for an extent. There are two local groups, each with 6 data fragments, plus 2 local parities (1 per group), and then 2 global parities. When using 9 upgrade domains, we put the 2 global parities into two upgrade domains with no other fragments in them, we then put the 2 local parities into the same upgrade domain with no other fragments in them, and then the remaining 6 upgrade domains hold the 12 data fragments for the 2 local groups.

During an upgrade period, when one upgrade domain is taken offline, every single data fragment can still be accessed efficiently - either reading directly from the fragment or reconstructing from other fragments within its local group.

4.4 Designing for Erasure Coding

Scheduling of Various I/O Types. The stream layer handles a large mix of I/O types at a given time: on-demand open/close, read, and append operations from clients, create, delete, replicate, reconstruct, scrub, and move operations generated by the system itself, and more. Letting all these I/Os happen at their own pace can quickly render the system unusable. To make the system fair and responsive, operations are subject to throttling and scheduling at all levels of the storage system. Every EN keeps track of its load at the network ports and on individual disks to decide to accept, reject, or delay I/O requests. Similarly, the SM keeps track of data replication load on individual ENs and the system as a whole to make decisions on when to initiate replication, erasure

coding, deletion, and various other system maintenance operations. Because both erasure coding and decoding requires accessing multiple ENs, efficiently scheduling and throttling these operations is crucial to have fair performance for other I/O types. In addition, it is also important to make sure erasure coding is keeping up with the incoming data rate from customers as well as internal system functions such as garbage collection. We have a Petabyte of new data being stored to WAS every couple of days, and the built out capacity expects a certain fraction of this data to be erasure-coded. Therefore, the erasure coding needs to be scheduled such that it keeps up with the incoming rate of data, even when there are critical re-replications that also need to be scheduled due to a lost disk, node or rack.

Reconstruction Read-ahead and Caching. Reconstruction of unavailable fragments is done in unit sizes greater than the individual append blocks (up to 5MB) to reduce the number of disk and network I/Os. This read-ahead data is cached in memory (up to 256MB) of the EN that has done the reconstruction. Further sequential reads are satisfied directly from memory.

Consistency of Coded Data. Data corruption can happen throughout the storage stack for numerous reasons [36]. In a large-scale distributed storage system, data can become corrupted while at rest, while being read or written in memory, and while passing through several data paths. Therefore, it is essential to check the consistency of the data in every step of the storage system operations in addition to periodically scrubbing the data at rest.

Checksum and parity are the two primary mechanisms to protect against data corruption [35]. In WAS, we employ various CRC (Cyclic Redundancy Check) fields to detect data and metadata corruptions. For example, each append block contains a header with CRC of the data block, which is checked when the data is written and every time data is read. When a particular data read or reconstruction operation fails due to CRC checks, the operation is retried using other combinations of erasure-coded fragments. Also, the fragment with the corrupted block is scheduled for regeneration on the next available EN.

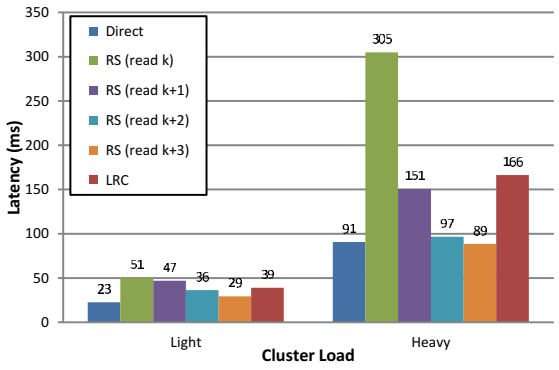
After each erasure encoding operation, several decoding combinations are tried from memory on the coordinator EN to check for successful restorations. This step is to ensure that the erasure coding algorithm itself does not introduce data inconsistency. For LRC (12, 2, 2), we perform the following decoding validations before allowing the EC to complete: i) randomly choose one data fragment in each local group and reconstruct it using its local group; ii) randomly choose one data fragment and reconstruct it using one global parity and the remaining data fragments; iii) randomly choose one data fragment and reconstruct it using the other global parity and the remaining data fragments; iv) randomly choose two data fragments and reconstruct them; v) randomly choose three data fragments and reconstruct them; and vi) randomly choose four data fragments (at least one in each group) and reconstruct them. After each decoding combination above, the CRC of the decoded fragment is checked against the CRC of the data fragment for successful reconstruction of data.

Finally, the coordinator EN performs a CRC of all of the final data fragments and checks that CRC against the original CRC of the full extent that needed to be erasure-coded. This last step ensures we have not used data that might become corrupted in memory during coding operations. If all these checks pass, the resulting coded fragments are persisted on storage disks. If any failure is detected during this process, the erasure coding operation is aborted, leaving the full extent copies intact, and the SM schedules erasure coding again on another EN later.

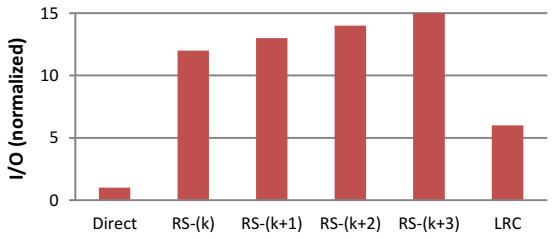
Arithmetic for Erasure Coding. Directly using Galois Field arithmetic for the erasure coding implementation is expensive because of all the emulation operations required on top of the integer arithmetic. Therefore, it is general practice to optimize Galois Field arithmetic operations by pre-computing and using addition and multiplication tables, which improves coding speed. In addition, Reed-Solomon codes can be further optimized by a transformation that enables the use of XOR operations exclusively [22]. To get the best possible performance for this transformation, the XOR operations can be ordered specifically based on the patterns in the coding and decoding matrices [23]. This scheduling removes most of the redundant operations and eliminates checking the coding matrix again and again during the actual coding pass. WAS uses all of the above optimizations to streamline in-memory encode and decode operations. Since modern CPUs perform XOR operations extremely fast, in-memory encode and decode can be performed at the speed which the input and output buffers can be accessed.

5 Performance

WAS provides cloud storage in the form of Blobs (user files), Tables (structured storage), Queues (message delivery), and Drives (network mounted VHDs). Applications have different workloads, which access each type of storage differently. The size of I/O can be polarized: small I/O is typically in the 4KB to 64KB range, predominantly assessing Tables and Queues; large I/Os (mostly 4MB), primarily accessing Blobs; and Drives can see a mixture of both. In this section, we characterize the performance of LRC and compare it to Reed-Solomon for small and large I/Os, respectively.

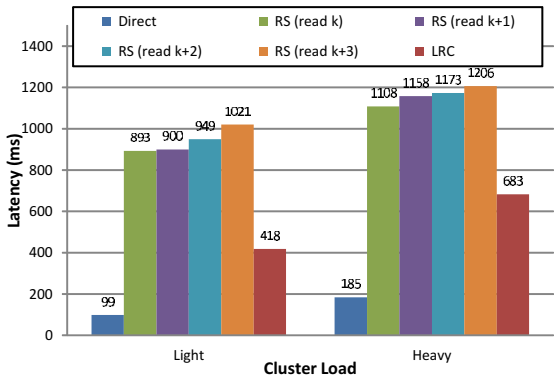


(a) Latency

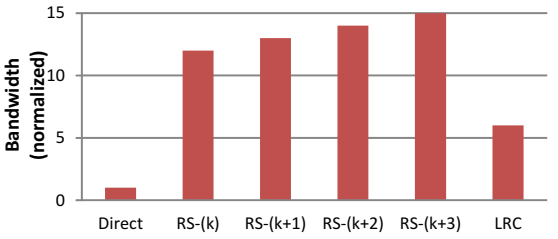


(b) Reconstruction I/O

Figure 9: **Small (4KB) I/O Reconstruction** - (12, 4) Reed-Solomon vs. (12, 2, 2) LRC.



(a) Latency



(b) Reconstruction Bandwidth

Figure 10: **Large (4MB) I/O Reconstruction** - (12, 4) Reed-Solomon vs. (12, 2, 2) LRC.

We compare LRC (12, 2, 2) to Reed-Solomon (12, 4), both of which yield storage cost at 1.33x. Note that (12, 3) Reed-Solomon is *not* an option, because its reliability is lower than 3-replication. Results are obtained on our production cluster with significant load variation over time. We separate the results based on the cluster load and contrast the gain of LRC when the cluster is lightly loaded to when it is heavily loaded. The production cluster, where these results were gathered, has one 1Gbps NIC for each storage node.

5.1 Small I/Os

The key metric for small I/O is latency and the number of I/Os taken by the requests. We run experiments with a mixture of direct read (reading a single fragment), reconstruction read with Reed-Solomon and LRC. The average latency results are summarized in Figure 9.

When the cluster load is light, all the latencies appear very low and comparable. There is not much difference between direct read and reconstruction with either Reed-Solomon or LRC. However, when the cluster load is heavy, there are definitely differences.

When a data fragment is unavailable, the reconstruction read with Reed-Solomon can be served by randomly selecting $k = 12$ fragments out of the remaining 15 fragments to perform erasure decoding. Unfortunately, the latency turns out much larger than that of direct read – 305ms vs. 91ms – because it is determined by the slowest fragment among the entire selection. Given the high latency, we exploit a simple technique - selecting more (denoted as k') fragments and decoding from the first k arrivals (represented as `RS (read k') or RS+(k')` in Figure 9). This technique appears very effective in weeding out slow fragments and reduces the latency dramatically.

In comparison, reconstruction with LRC is fast. It requires only 6 fragments to be read and achieves the latency of 166ms, which is comparable to 151ms with Reed-Solomon reading 13 fragments. Note that extremely aggressive reads (reading all 15) with Reed-Solomon achieves even lower latency, but it comes at a cost of many more I/Os. The relative number of I/Os, normalized by that of direct read, is shown in Figure 9(b). The fast reconstruction and the I/O cost savings are the reasons why we chose LRC over Reed-Solomon for Windows Azure Storage's erasure coding.

5.2 Large I/Os

We now examine the reconstruction performance of 4MB large I/Os. The key metric is latency and bandwidth consumption. We compare direct read to reconstruction with Reed-Solomon and LRC. The results are presented in Figure 10.

The results are very different from the small I/O case. Even when the cluster load is light, the reconstruction with erasure coding is already much slower than direct read, given the amount of bandwidth it consumes. Compared to direct read taking 99ms, Reed-Solomon with 12

fragments takes 893ms – 9 times slower.

In large I/Os, the latency is mostly bottlenecked by network and disk bandwidth, and the bottleneck for these results was the 1Gbps network card on the storage nodes. Since LRC reduces the number of fragments by half, its latency is 418ms and significantly reduced from that of Reed-Solomon. Note that, different from the small I/O case, aggressive reads with Reed-Solomon using more fragments does not help, but rather hurts latency. The observation is similar when the cluster load is heavy.

Because of the significantly reduced latency and the bandwidth savings, which are particularly important when the system is under heavy load or has to recover from a rack failure, we chose LRC for Windows Azure Storage.

5.3 Decoding Latency

To conclude the results, we also wanted to compare the latency spent on decoding fragments between Reed-Solomon and LRC. The average latency of decoding 4KB fragments is 13.2us for Reed-Solomon and 7.12us for LRC. Decoding is faster in LRC than Reed-Solomon because only half the number of fragments are involved. Even so, the decoding latencies are typically in microseconds and *several orders of magnitude smaller* than the overall latency to transfer the fragments to perform the reconstruction. Therefore, from the latency of decoding standpoint, LRC and Reed-Solomon are comparable. Note that, pure XOR-based codes, such as Weaver codes [18], HoVer codes [19] and Stepped Combination codes [20], can be decoded even faster. The gain of faster decoding, however, would not matter in WAS, as the decoding time is orders of magnitude smaller than the transfer time.

6 Related Work

Erasure Coding in Storage Systems: Erasure coding has been applied in many large-scale distributed storage systems, including storage systems at Facebook and Google [3, 4, 5, 8, 9, 10]. The advantage of erasure coding over simple replication is that it can achieve much higher reliability with the same storage, or it requires much lower storage for the same reliability [7]. The existing systems, however, do not explore alternative erasure coding designs other than Reed-Solomon codes [13]. In this work, we show that, under the same reliability requirement, LRC allows a much more efficient cost and performance trade-off than Reed-Solomon.

Performance: In erasure-coded storage systems, node failures trigger rebuilding process, which in turn results in degraded latency performance on reconstruction reads [6]. Moreover, experience shows that transient errors in which no data are lost account for more than 90% of data center failures [10]. During these periods as well as upgrades, even though there is no background data rebuilding, reads trying to access unavailable nodes are still served through reconstruction.

Complementary to system techniques, such as load balancing and prioritization [11], LRC explores whether the erasure coding scheme itself can be optimized to reduce repair traffic and improve user I/Os.

Erasure Code Design: LRC is a critical improvement over our own Pyramid codes [14]. LRC exploits non-uniform parity degrees, where some parities connect to fewer data nodes than others. Intuitively, the parities with fewer degrees facilitate efficient reconstruction. This direction was originally pioneered for communication by landmark papers on Low Density Parity Check (LDPC) codes [15, 16]. LDPC were recently explored in the area of storage [17, 20]. In particular, Plank et al. [17] applied enumeration and heuristic methods to search for parity-check erasure codes of small length. Due to exponential search space, the exploration was limited to 3, 4 and 5 parities. The codes discovered cannot tolerate arbitrary three failures, which is the minimum requirement in WAS. Stepped Combination codes [20] are LDPC codes with very small length, offering fault tolerance guarantee and efficient reconstruction, but do not provide the same trade-offs that LRC can achieve..

Reed-Solomon Codes are Maximum Distance Separable (MDS) codes [12], which require minimum storage overhead for given fault tolerance. LRC is not MDS and thus requires higher storage overhead for the same fault tolerance. The additional storage overhead from the local parities are exploited for efficient reconstruction. This direction of trading storage overhead for reconstruction efficiency is also explored by other state-of-the-art erasure codes designed specifically for storage systems, such as Weaver codes [18], HoVer codes [19] and Stepped Combination codes [20]. We show that LRC achieves better trade-offs than these modern storage codes for WAS' erasure coding design goals .

To improve reconstruction performance, instead of reading from fewer fragments as in LRC, a promising alternative is to read instead from more fragments, but less data from each [24, 25, 26, 27]. However, practical solutions known so far [26, 27] achieve only around 20%-30% savings in terms of I/O and bandwidth, much less than LRC.

7 Summary

Erasure coding is critical to reduce the cost of cloud storage, where our target storage overhead is 1.33x of the original data. When using erasure coding, fast reconstruction of offline data fragments is important for performance. In Windows Azure Storage, these data fragments can be offline due to disk, node, rack and switch failures, as well as during upgrades.

We introduced Local Reconstruction Codes as a way to reduce the number of fragments that need to be read from to perform this reconstruction, and compared LRC to Reed-Solomon. We showed that LRC (12, 2, 2), which has a storage overhead of 1.33x, saves significant I/Os and bandwidth during reconstruction when compared to Reed-Solomon (12, 4). In terms of latency, LRC has comparable latency for small I/Os and better latency for large I/Os.

We chose LRC (12, 2, 2) since it achieves our 1.33x storage overhead target and has the above latency, I/O and bandwidth advantages over Reed-Solomon. In addition, we needed to maintain durability at the same or higher level than traditional 3 replicas, and LRC (12, 2, 2) provides better durability than the traditional approach of keeping 3 copies. Finally, we explained how erasure coding is implemented, some of the design considerations, and how we can efficiently lay out LRC (12, 2, 2) across the 20 fault domains and 10 upgrade domains used in Windows Azure Storage.

8 Acknowledgements

We would like to thank Andreas Haeberlen, Geoff Voelker, and anonymous reviewers for providing valuable feedback on this paper. We would also like to thank all of the members of the Windows Azure Storage team.

References

[1] B. Calder et al., "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency," *ACM SOSP*, 2011.

[2] D. T. Meyer et al., "Fast and Cautious Evolution of Cloud Storage," *HotStorage*, 2010.

[3] J. Kubiatowicz et al., "OceanStore: An Architecture for Global-Scale Persistent Storage," *ACM ASPLOS*, Nov. 2000.

[4] A. Haeberlen, A. Mislove, and P. Druschel, "Glacier: Highly durable, decentralized storage despite massive correlated failures," *USENIX NSDI*, 2005.

[5] M. Abd-El-Malek et al., "Ursa Minor: Versatile Cluster-based Storage," *USENIX FAST*, 2005.

[6] C. Ungureanu et al., "HydraFS: A High-Throughput File System for the HYDRAstor Content-Addressable Storage System," *USENIX FAST*, 2010.

[7] H. Weatherspoon, and J. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," *In Proc. IPTPS*, 2001.

[8] D. Borthakur et al., "HDFS RAID," *Hadoop User Group Meeting*, Nov. 2010.

[9] A. Fikes, "Storage Architecture and Challenges," *Google Faculty Summit*, 2010.

[10] D. Ford et al., "Availability in Globally Distributed Storage Systems,", *USENIX OSDI*, 2010.

[11] L. Tian et al., "PRO: A Popularity-based Multi-threaded Reconstruction Optimization for RAID-Structured Storage Systems," *USENIX FAST*, 2007

[12] F. J. MacWilliams and N. J. A. Sloane, "The Theory of Error Correcting Codes," *Amsterdam: North-Holland*, 1977.

[13] I. S. Reed and G. Solomon, "Polynomial Codes over Certain Finite Fields", *J. SIAM*, 8(10), 300-304, 1960.

[14] C. Huang, M. Chen, and J. Li, "Pyramid Codes: Flexible Schemes to Trade Space for Access Efficiency in Reliable Data Storage Systems," *Proc. of IEEE NCA*, Cambridge, MA, Jul. 2007.

[15] R. G. Gallager, "Low-Density Parity-Check Codes," *MIT Press*, Cambridge, MA, 1963.

[16] M. G. Luby et al., "Efficient Erasure Correcting Codes," *IEEE Transactions on Information Theory*, 2011.

[17] J. S. Plank, R. L. Collins, A. L. Buchsbaum, and M. G. Thomason, "Small Parity-Check Erasure Codes - Exploration and Observations," *Proc. DSN*, 2005.

[18] J. L. Hafner, "Weaver codes: Highly fault tolerant erasure codes for storage systems," *USENIX FAST*, 2005.

[19] J. L. Hafner, "HoVer Erasure Codes for Disk Arrays," *Proc. of DSN*, 2006.

[20] K. M. Greenan, X. Li, and J. J. Wylie, "Flat XOR-based erasure codes in storage systems: Constructions, efficient recovery, and tradeoffs," *IEEE Mass Storage Systems and Technologies*, 2010.

[21] P. Gopalan, C. Huang, H. Simitci, and S. Yekhanin, "On the Locality of Codeword Symbols," *Allerton*, 2011.

[22] J. Blomer et al., "An XOR-Based Erasure-Resilient Coding Scheme," Technical Report No. TR-95-048, ICSI, Berkeley, California, Aug. 1995.

[23] J. Luo, L. Xu, and J. S. Plank, "An Efficient XOR-Scheduling Algorithm for Erasure Codes Encoding," *Proc. DSN*, Lisbon, Portugal, June, 2009.

[24] A. G. Dimakis et al., "Network Coding for Distributed Storage Systems," *IEEE Transactions on Information Theory*, Vol. 56, Issue 9, Sept. 2010.

[25] L. Xiang et al., "Optimal recovery of single disk failure in RDP code storage systems," *ACM SIGMETRICS*, 2010.

[26] O. Khan et al., "Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads," *USENIX FAST*, San Jose, Feb. 2012.

[27] Y. Hu et al., "NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-Clouds," *USENIX FAST*, San Jose, 2012.

[28] J. H. Howard et al., "Scale and Performance in a Distributed File System," *ACM ToCS*, Feb. 1988.

[29] M. Satyanarayanan et al., "CODA: A Highly Available File System for a Distributed Workstation Environment," *IEEE Transactions on Computers*, Apr. 1990.

[30] B. Liskov et al., "Replication in the Harp File System," *ACM SOSP*, Pacific Grove, CA, Oct. 1991.

[31] F. Dabek et al., "Wide-Area Cooperative Storage with CFS," *ACM SOSP*, 2001.

[32] B. G. Chun et al., "Efficient Replica Maintenance for Distributed Storage Systems," *USENIX NSDI*, 2006.

[33] Q. Xin et al., "Reliability mechanisms for very large storage systems," *Proc. of IEEE Conference on Mass Storage Systems and Technologies*, 2003.

[34] S. Nath et al., "Subtleties in Tolerating Correlated Failures in Wide-Area Storage Systems," *USENIX NSDI*, 2006.

[35] A. Krioukov et al., "Parity Lost and Parity Regained," *USENIX FAST*, Feb. 2008.

[36] L. N. Bairavasundaram et al., "An Analysis of Data Corruption in the Storage Stack," *USENIX FAST*, Feb. 2008.

[37] L. Lamport, "The Part-Time Parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133-169, May 1998.

[38] J. K. Resch, and J. S. Plank, "AONT-RS: Blending Security and Performance in Dispersed Storage Systems," *USENIX FAST*, Feb. 2011.

Composable Reliability for Asynchronous Systems

Sunghwan Yoo[1,2] Charles Killian[1] Terence Kelly[2] Hyoun Kyu Cho[2,3] Steven Plite[1]
[1]*Purdue University* [2]*HP Labs* [3]*University of Michigan*

Abstract

Distributed systems designs often employ replication to solve two different kinds of availability problems. First, to prevent the loss of data through the permanent destruction or disconnection of a distributed node, and second, to allow prompt retrieval of data when some distributed nodes respond slowly. For simplicity, many systems further handle crash-restart failures and timeouts by treating them as a permanent disconnection followed by the birth of a new node, relying on peer replication rather than persistent storage to preserve data. We posit that for applications deployed in modern managed infrastructures, delays are typically transient and failed processes and machines are likely to be restarted promptly, so it is often desirable to resume crashed processes from persistent checkpoints. In this paper we present MaceKen, a synthesis of complementary techniques including Ken, a lightweight and decentralized rollback-recovery protocol that transparently masks crash-restart failures by careful handling of messages and state checkpoints; and Mace, a programming toolkit supporting development of distributed applications and application-specific availability via replication. MaceKen requires near-zero additional developer effort—systems implemented in Mace can immediately benefit from the Ken protocol by virtue of following the Mace execution model. Moreover, this model allows multiple, independently developed application components to be seamlessly composed, preserving strong global reliability guarantees. Our implementation is available as open source software.

1 Introduction

Our work matches failure handling in distributed applications to deployment environments. In managed infrastructures, unlike the broader Internet, crash-restart failures are common relative to permanent-departure failures. Moreover, correlated failures are more likely: Application nodes are physically co-located, increasing their susceptibility to simultaneous environmental failures such as power outages; routine maintenance will furthermore restart machines either simultaneously or sequentially. Our toolkit masks crash-restart failures, preventing such brief or correlated failures from causing data loss or increased protocol overhead due to application-level failure handling.

Traditional wide-area distributed systems employ replication to solve two different kinds of availability problems. First, to prevent the loss of data through the permanent destruction or disconnection of a node, and second, to allow prompt data retrieval when some nodes respond slowly. Persistent storage can protect data from crash-restart failures, but it must be handled very carefully to avoid replica consistency problems or data corruption. For example, recovering a key-value store node requires checking data integrity and freshness and forwarding data to the new nodes responsible for it if the mapping has changed. Recovery can be quite tricky, particularly as little is known of the disk and network I/O in progress when the failure occurred. Recovery is further complicated if multiple independently developed distributed systems interact. Given that replication will be used anyway to ensure availability, and because correctly recovering persistent data after failures is difficult, many distributed systems choose to handle crash-restart failures and timeouts by treating them as a permanent disconnection followed by the birth of a new node, relying on peer replication, rather than persistent storage, to preserve data.

As new applications are increasingly deployed in managed environments, one appealing approach is to deploy wide-area distributed systems directly in these managed environments. However, without persistent storage, a simultaneous failure of all nodes (e.g., a power outage) would destroy all data. A more modest failure scenario in which machines are restarted sequentially for maintenance may be acceptable for a distributed system that does not employ persistent storage, but only if the system can process churn and update peer replicas quickly enough that all copies of any individual datum are not simultaneously destroyed. Wide-area distributed systems, such as P2P systems, are therefore often not well-suited to tolerate the types of failures more likely to occur in managed infrastructures, despite being designed to tolerate a high rate of *uncorrelated* failures. If crash-restart failures can be masked, however, such systems can ignore challenging correlated failures while still providing replication-based availability. Additionally, as node departures are infrequent, and performance less variable across managed nodes, in some cases fewer replicas will suffice to meet availability requirements.

At the other end of the spectrum, some applications that run in managed infrastructures do not require strong

availability—distributed batch-scientific computing applications can seldom afford replication because they often operate at the limits of available system memory. If a failure occurs in these applications, they wish to lose as little time to re-computation as possible. In the worst case, a computation can be restarted from the input data. If we can mask crash-restart failures, we remove a large class of possible failures for distributed batch computing applications in managed clusters, as the cluster machines are unlikely to fail permanently during any given batch run.

In this paper, we describe the design and implementation of Ken, a protocol that transforms integrity-threatening crash-restart failures into performance problems, even across independently developed systems (Section 3). Ken uses a lightweight, decentralized, and uncoordinated approach to checkpoint process state and guarantee *global* correctness. Our benchmarks demonstrate that Ken is practical under modest assumptions and that non-volatile memory will improve its performance substantially (Section 5.1).

We further explain how Ken is a perfect match for a broad class of event-driven programming techniques, and we describe the near-transparent integration of Ken and the Mace [20] toolkit, yielding MaceKen (Section 4). Systems developed for Mace can be run using MaceKen with little or no additional effort. We evaluate MaceKen in both distributed batch-computing environments and a distributed hash table (Sections 5.2 and 5.3), demonstrating how Ken enables unmodified systems to tolerate otherwise debilitating failures. We also developed a novel technique to accurately emulate host failures using Linux containers [26]—basic process death, e.g., from killing the process, causes socket failures to be promptly reported to remote endpoints, which does not occur in power failures or kernel panics. We show how enabling Ken for a system developed for the Internet can prepare it for deployment in managed environments susceptible to correlated crash-restart failures. Existing application logic to route around slow nodes will continue to address unresponsive nodes, while safely remaining oblivious to quick process restarts.

Finally, we illustrate a broader, fundamental contribution of the Ken protocol: the effortless composition of independently developed systems and services, retaining the same reliability guarantees when the systems interact with each other without coordination, even during failure. In this example, a hypothetical scenario involving auctions and banking, failures would normally lead to loss of money or loss of trade. Ken avoids all such problems under heavy injected failures (Section 5.4).

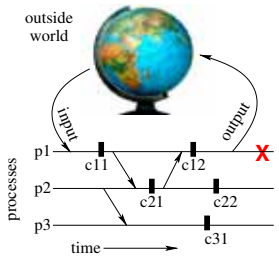


Figure 1: Abstract distributed computation

2 Background

Before describing the Ken protocol, we first review relevant concepts surrounding fault-tolerant distributed computing. Ken allows the developer to treat failed processes and hosts simply as slowly responding nodes, even across independently developed systems. We explain how Ken provides *distributed consistency*, *output validity*, and *composable reliability*.

To understand these concepts, consider Figure 1, illustrating standard concepts of distributed computing [12]. In the figure, time advances from left to right. Distributed computing processes p_1, p_2, and p_3 are represented by horizontal lines. Processes can exchange messages with each other, represented by diagonal arrows, and take checkpoints of their local state, represented by black rectangles. A crash, represented by a red "X," destroys process state, which may be restored from a checkpoint previously taken by the crashed process. Processes may also receive inputs from, and emit outputs to, the outside world. The outside world differs from the processes in two crucial ways: It cannot replay inputs, nor can it roll back its state. Therefore inputs are at risk of loss before being checkpointed and outputs are irrevocable.

2.1 Distributed Consistency

Checkpoints taken by two different processes are termed *inconsistent* if one records receiving a message the other does not record sending, because the message was sent *after* the sender's checkpoint was taken. Checkpoints c_{11} and c_{21} in the figure are inconsistent because c_{21} records the receipt of the message from p_1 to p_2 but c_{11} does not record having sent it. A set of checkpoints, one per process, is called a *recovery line* if no pair of checkpoints is inconsistent. A recovery line represents a sane state to which the system may safely be restored in response to failure. A major challenge in building durable systems lies in the efficient maintenance of recovery lines. If a process were simply to take checkpoints at fixed time intervals, some may not be suitable for any recovery line. A checkpoint is termed *useless* if it cannot legally be part

of any recovery line. Checkpoint c_{21} is useless, as it is inconsistent with both checkpoints c_{11} and c_{12}.

One of the best-known approaches to constructing recovery lines is the Lamport-Chandy algorithm [5]. This algorithm requires distributed coordination, adding overheads, especially if checkpoints are taken frequently. Additionally, coordinated checkpoints may be impractical if independently developed/deployed applications are composed as discussed in Section 2.3.

2.2 Output Validity

Outputs emitted to the outside world raise special difficulties. Because the outside world by definition cannot roll back its state, we must assume that it "remembers" all outputs externalized to it. Therefore the latter may not be "forgotten" by the distributed system that emitted them, lest inconsistency arise. Distributed systems must obey the *output commit* rule: All externalized outputs must be recorded in a recovery line. In Figure 1, the output by process p_1 violates the output commit rule, and the subsequent crash causes the system to forget having emitted an irrevocable output.

Failures (crashes and message losses) may disturb a distributed computation. We say that a distributed system satisfies the property of *output validity* if the sequence of outputs that it emits to the outside world *could have been* generated by failure-free operation. Lowell et al. discuss closely related concepts in depth [25].

2.3 Composable Reliability

Even if individual applications support distributed consistency and output validity, these properties need not apply to the *union* of the applications when the latter interact. Composing together independently developed and independently deployed/managed applications is very common in practice. In such scenarios, the global guarantees of distributed consistency and output validity require maintaining a recovery line spanning multiple independently developed applications, coordinating rollback across independently managed systems to reach a globally-consistent recovery line, and globally enforcing the output commit rule across administrative domains. We show that Ken provides a *local* solution, maintaining recovery lines, enforcing output commit, and recovering from failures without cross-application coordination.

3 Reliability Mechanism

Below we describe the Ken protocol as we have implemented it, its programming model, and its properties. The name and the essence of the protocol are taken from Waterken, an earlier Java distributed platform that presents different programming abstractions [6, 17].

3.1 Protocol

Ken processes exchange discrete, bounded-length messages with one another and interact with the outside world by receiving inputs and emitting outputs. Incoming messages/inputs trigger computations with two kinds of consequences: outbound messages/outputs, and local state changes. Each Ken process contains a single input-handling loop, an iteration of which is called a *turn*.

Ken turns are *transactional*: either all of their consequences are fully realized, or else it is as though the message or input that triggered the turn never arrived. During a turn, outbound messages and outputs are buffered locally rather than being transmitted. At the end of a turn all such messages/outputs and local state changes caused by the turn are atomically committed to durable storage. On checkpoint success, the buffered messages/outputs become eligible for transmission; otherwise they are discarded and process state is rolled back to the start of the turn. The Ken protocol does not prescribe a storage medium; implementation-specific requirements of disaster tolerance, monetary cost, size, speed, density, power consumption, and other factors may guide the choice of storage. Ken simply requires the ability to recover intact all checkpointed data following any tolerated failure.

Messages from successful turns are re-transmitted until acknowledged. An acknowledgment indicates that the recipient has not only *received* the message but has also *processed it to completion*. The ACK assures the sender that the turn triggered by the message ended well, i.e., all of its consequences were fully realized and atomically committed. The sender may therefore cease re-transmitting ACK'd messages and delete them from durable storage. Message sequence numbers ensure FIFO delivery between each sender-receiver pair and ensure that each message is processed exactly once. Outside-world interactions may have weaker semantics than messages exchanged among the "inside world" of protocol-compliant Ken processes, because by definition the outside world cannot be relied upon to replay inputs or acknowledge outputs. Crashes may destroy an input upon arrival, and may destroy evidence of a successful output a moment after such evidence is created. Specific input and output devices and corresponding drivers that mediate outside-world interactions may be able to offer stronger guarantees than at-most-once input processing and at-least-once output externalization, depending on the details of the devices concerned [17]. Our Ken implementation allows drivers to communicate with a Ken process via `stdin` and `stdout`.

Recovery in Ken is straightforward. Crashes destroy the contents of local volatile memory. Recovery consists of restoring local state from the most recent checkpoint and resuming re-transmission of messages from successfully completed turns. Recovery is a purely local affair and does not involve any interaction with other Ken processes nor any message/input/event replay. Because Ken's transactional turns externalize their full consequences if and only if they complete successfully, a Ken process that crashes and recovers is indistinguishable from one that is merely slow.

Two sources of nondeterminism may affect Ken computations: *local* nondeterminism in the hardware and software beneath Ken's event loop, and nondeterminism in the *interleaving of messages* from several senders at a single receiver. Ken ignores both. A crash may therefore change output from what it would have been had the crash not occurred. Consider a turn that intends to output the local time but crashes before the turn completes. Following recovery, the time will be emitted, but it will differ compared with failure-free behavior. Next, consider a Ken process that intends to concatenate incoming messages from multiple sources and output a checksum of the concatenation. The order in which messages arrive at this process from different senders may differ in a crash/recovery scenario versus failure-free operation; as a result the checksum output will also differ. In both cases, crashes result in outputs that are *different* but not *unacceptable* compared with failure-free outputs. As there exists a hypothetical failure-free execution with the same outputs, output validity holds.

Two further examples illustrate how Ken's approach to nondeterminism is sometimes positively beneficial. First, consider an overflow-intolerant "accumulator" process that accepts signed integers as messages, and adds them to a counter, initially zero. If three messages containing `INT_MAX`, `1`, and `INT_MIN` arrive from different senders in that order, the `1` will crash the accumulator. Following recovery, the re-transmitted `1` may arrive *after* `INT_MIN`, averting overflow. Next, consider a Ken-based "square root server." Requests containing 4, 9, and 25 elicit replies of 2, 3, and 5 respectively. Unfortunately the server is unimaginative—e.g., it crashes when asked to compute $\sqrt{-1}$. Requests containing perfect squares, however, will continue to be served correctly whenever they reach the server between crashes caused by undigestible requests; mishandled requests impair performance but do not cause incorrect replies to acceptable requests. Wagner calls this guarantee *defensive consistency* [10]. Ken ensures defensive consistency provided that bad inputs crash turns *before* they complete (e.g., via assertion failures). Our simple examples represent the kinds of corner-case inputs and "Heisenbugs" that commonly cause problems in practice. Ken

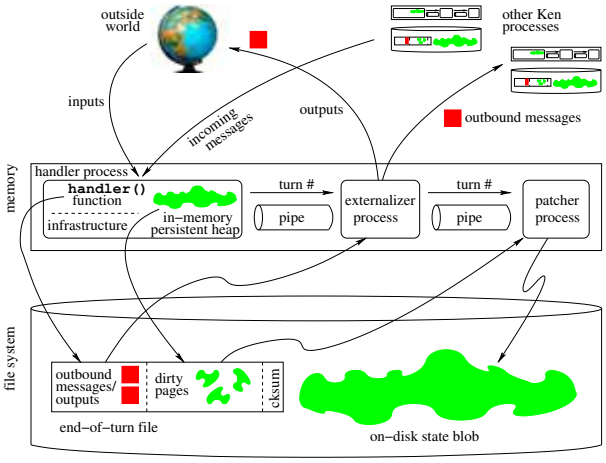


Figure 2: Ken internals

sometimes allows naturally occurring nondeterminism to work in our favor: forgiving recovery with zero programmer effort is a natural side effect of abandoning deterministic replay as a goal. See Lowell et al. for a detailed discussion of the potentials and limitations of approaches that leverage nondeterminism to "erase" failures [25].

3.2 Implementation

Implementing generic support infrastructure for transactional event loops requires factoring out several difficult problems that would otherwise need to be solved by individual applications, e.g., efficient incremental checkpointing and reliable messaging. Furthermore it is not enough merely to provide these generic facilities separately; they must be carefully *integrated* to provide Ken's strong global correctness guarantees (distributed consistency and output validity). We describe first the programming model then the internal details of our Ken implementation in C for POSIX-compliant Unix systems such as Linux. Figure 2 illustrates the basic components and their flow of data.

Ken supports an event-driven programming paradigm familiar to many systems programmers and, thanks to JavaScript/AJAX, many more application programmers [29]. Whereas a conventional C program defines a `main()` function executed whenever the program is invoked, a Ken program defines a `handler()` function called by the Ken infrastructure in response to inputs, messages from other Ken processes, and alarm expirations. The handler may `send()` messages to other Ken processes, identified as network addresses, or emit outputs by specifying "`stdout`" as the destination in a `send()` call. The handler may also manipulate a persistent heap via `ken_alloc()` and `ken_free()` functions analogous to their conventional counterparts. The handler must eventually return (versus loop infinitely), and it

may specify via its integer return value a time at which it should be invoked again if no messages or inputs arrive. A return value of -1 indicates there is no such timeout. An API allows application software to determine whether a given message has been acknowledged.

The Ken infrastructure contains the event loop that calls the application-level `handler()` function, passing inputs/messages as arguments. The sender of the message is also passed as an argument; in the case of inputs, the sender is "`stdin`." As the handler executes, the infrastructure appends outbound messages from `send()` calls to an end-of-turn (EOT) file whose filename contains the turn number. The infrastructure also manages the Ken persistent heap, tracking which memory pages have been modified: At the start of every turn the Ken heap is read-only. The first STORE to a memory page generates a segmentation fault; Ken catches the SIGSEGV, notes the page, and makes it writable. When the handler function returns, the infrastructure appends the turn's dirty pages to the EOT file along with appropriate metadata. Finally, Ken appends a 32-bit checksum to the EOT file.

As illustrated in Figure 2, a logical Ken process consists of three Unix processes: The *handler* process contains both the application-level handler function and most of the Ken infrastructure. The *externalizer* process re-transmits outbound messages until they are acknowledged. The *patcher* process merges dirty pages from EOT files into the *state blob* file, which contains the Ken persistent heap plus a few pages of metadata.

When the handler process concludes a turn, it sends the turn number to the externalizer via a pipe. The externalizer responds by `fsync()`ing both the EOT file and its parent directory, which commits the turn and allows the EOT file's messages/outputs to be externalized and its dirty pages to be patched into the state blob file; it also allows the incoming message that started the turn to be acknowledged. The externalizer writes outputs to `stdout` and transmits messages to their destinations in UDP datagrams. Messages to Ken processes are re-transmitted until acknowledged.

The externalizer tells the patcher a turn concluded successfully by writing the turn number to a second pipe. The patcher considers EOT files in turn order, pasting the dirtied pages into the state blob file at the appropriate offsets then `fsync()`ing the state blob. When all pages in an EOT file have been incorporated, and all messages in the EOT file have been acknowledged, the EOT file is deleted. As the patching process is idempotent, crashes during patching are tolerated and any state blob corruption caused by such crashes is repaired upon recovery.

Ken's three-Unix-process design complicates the implementation somewhat, but it carries several benefits. It decouples the handling of incoming messages, which generates EOT files, from the processing and deletion of EOT files. The `fsync()`s required to ensure durability occur in parallel with execution of the next turn, because the former are performed by the externalizer process and the latter in the handler process. If the handler process generates EOT files faster than the externalizer and patcher can consume them, the pipes containing completed turn numbers eventually fill, causing the handler process to block until the externalizer and patcher processes catch up.

Resurrecting a crashed Ken process begins by ensuring that all three of the Unix processes constituting its former incarnation are dead. A simple shell script suffices to detect a crash, thoroughly kill the failed Ken process, and restart it.[1] Ken's recovery code typically discovers that the most recent EOT file does not contain a valid checksum; the file is then deleted. The dirty pages in remaining EOT files are patched into the state blob file, which is then `mmap()`'d into the address space of the reincarnated handler process. We rely on `mmap()` to place the state blob at an exact address, otherwise pointers within the persistent heap would be meaningless. POSIX does not guarantee that `mmap()` will honor a placement address hint, but Linux and HP-UX kernel developers confirm that both OSes always honor the hint. In our experience `mmap()` always behaves as required. The externalizer process of a recovered Ken process simply resumes the business of re-transmitting unacknowledged messages.

3.3 Programming Guidelines

Ken programmers observe a handful of guidelines that follow from the abstract protocol, and our current implementation imposes a few additional restrictions.

The most important guideline is easy to follow: Write code as though tolerated failures cannot occur. Application programs running atop Ken never need to handle such failures, nor should they attempt to infer them. The most flagrant violation of output validity would be a Ken process that counts the number of times that it crashed and outputs this information. To provide a safe outlet for debugging diagnostics, we treat the standard error stream as "out of band" and exempt from Ken rules. Developers may use `stderr` in the customary fashion, with a few caveats: The three Unix processes of a logical Ken process share the same `stderr` descriptor, and to prevent badly interleaved error messages Ken applications should `write()` rather than `fprintf()` to `stderr`; furthermore `stderr` should pass through a pipe before being redirected to a file. Most importantly, `stderr` is "write-

[1] A Ken process that wishes to terminate *permanently* may convey to the resurrection script a "do not resuscitate" order via, e.g., an exit code, after confirming that sent messages have been acknowledged; terminating earlier would break the basic model and void Ken's warranties.

only." All bets are off if information written to `stderr` in violation of Ken's turn discipline finds its way back into the system.

Experienced programmers typically resist the next rule initially, then gradually grow to appreciate it: Deliberately crashing a Ken program is always acceptable and sometimes recommended, e.g., when corruption occurs and is detected during a turn. Crashing returns local Ken process state to the start of the turn, before the corruption occurred. Note that Ken substantially relaxes the traditional fail-stop recommendation that applications should try to crash *as soon as possible* after bugs bite [25]. Ken programmers may safely postpone corruption detection to immediately before the `handler()` function returns, i.e., Ken allows invariant verification and corruption detection to be safely consolidated at a single well-defined point. Assertions provide manual, application-specific invariant verification. Correia et al. describe a *generic* and *automatic* complementary mechanism for catching corruption due to hardware failures, e.g., bit flips [7].

Crashing a Ken program can do more than merely undo corruption. Memory exhaustion provides a good illustration of how crashing a Ken process can solve the root cause of a problem: The virtual memory footprint of a Unix process is the number of memory pages dirtied during execution, and a Ken process is no exception. Unlike an ordinary Unix process, however, Ken effectively migrates data in the persistent heap to the file system as a side effect of crash recovery. Upon recovery the Ken persistent heap is stored in the state blob file and the resurrected handler process contains only a read-only mapping, requiring no RAM or swap [30]. Persistent heap data will be copied into the process's address space on demand. Cold data consumes space in the *file system* rather than RAM or swap, which are typically far less abundant than file system space. A Ken program that calls `assert(0)` when `ken_malloc()` returns `NULL` thereby *solves the underlying resource scarcity problem.*

Ken applications must conform to the transactional turn model. Handler functions that cause externally visible side effects, e.g., by calling legacy library functions that transmit messages under the hood *during* a turn, void Ken's warranties. Conventional writes to a conventional file system from the handler function similarly break the transactional turn model because a crash between writes visibly leaves ordinary files in an inconsistent state. The preferred Ken way to store data durably, of course, is to use the persistent heap, though a basic filesystem driver could be implemented using Ken inputs and outputs.

Static, external, and global variables should be avoided because they are not preserved across crashes; Ken provides alternative means of finding entry points into the persistent heap. For example, Ken includes a hash table interface to heap entry points that is nearly as convenient as the static and global variables it is often used to replace. The biggest problem in practice is legacy libraries that employ static storage, e.g., old-fashioned non-reentrant random number generators and the standard `strtok()` function. In most cases safe alternatives are available, e.g., `strtok_r()`. The conventional memory allocator should not be used because the conventional heap doesn't survive crashes. Ken novices should limit themselves to the Ken persistent heap; knowledgeable programmers might consider, e.g., using `alloca()` for intra-turn scratch space.

Multithreading within a turn is possible in principle, but not recommended because Ken currently does not automatically preserve thread stacks across crashes. One easy pattern is guaranteed to work: Threads spawned by the handler function terminate before it returns. Trickier patterns involving threads that persist across handler invocations require more careful programming. Much of our own work explores shared-nothing message-passing computation, which plays to Ken's strengths, and we are often able to avoid the use of threads altogether.

Ken supports reliable unidirectional "fire and forget" messages, not blocking RPCs. We have not implemented RPCs for several reasons. First, they can be susceptible to distributed circular-wait deadlock whereas unidirectional messages are not. Furthermore output commit requires checkpointing all relevant process state prior to externalizing an RPC request, and in this case relevant state would include the stack, making checkpoints larger. More importantly, RPCs would disrupt Ken's "transactional turn" semantics as they externalize a request during a turn. In our experience it is often natural and easy to design a distributed computation in an event-driven style based on reliable unidirectional messages. The popularity of event-driven frameworks such as AJAX suggests that programming without RPCs is widely applicable.

A final area that requires care is system configuration. Most importantly, data integrity primitives such as `fsync()` must ensure durability. Storage devices often contain volatile write caches that do not tolerate power failures, which must be disabled. On some systems a small number of UDP datagrams can fill the default socket send/receive buffers; configuring larger ceilings via the `sysctl` utility allows Ken to increase per-socket buffers via `setsockopt()`, which reduces the likelihood of datagram loss. Other system parameters that sometimes reward thoughtful tuning are those that govern memory overcommitment and the maximum number of memory mappings. Multiple Ken processes running on a single machine should be run in separate directories for better performance.

3.4 Properties

Ken turns impose atomic, isolated, and durable changes on application state. If the application-level handler function always leaves the persistent heap in a consistent state when it returns—hopefully the programmer's intention!—then Ken provides ACID transactions that ensure local application state integrity. Ken also guarantees reliable pairwise-FIFO messages with exactly-once consumption. These benefits accrue without any overt act by the programmer; reliability is transparent.

By contrast, a common pattern in existing commercial software for achieving both application state and message reliability is to use a relational database to ensure local data integrity and message-queuing middleware to ensure reliable communications. In the RDBMS/MQ pattern it is the programmer's responsibility to orchestrate the delicate interplay between transactions evolving application data from one consistent state to the next and operations ensuring message reliability. The slightest error can easily violate global correctness, e.g., by overlooking the output commit rule or allowing a crash to introduce distributed inconsistencies. Transparent reliability is valuable even for relatively simpler batch scientific programs, where experience has shown that even experts find it very difficult to insert appropriate checkpoints [1].

When used as directed, Ken makes it *impossible* for the programmer to compromise distributed consistency or output validity. Distributed consistency in Ken follows directly from the fact that Ken performs an output commit atomically with every message sent and every output. The set of most recent per-process checkpoints in a system of Ken processes always constitutes a recovery line. Output validity follows from the fact that failures (message losses and/or process crashes) put a system of Ken processes into a state that could have resulted instead from message delays. More formal discussions of distributed consistency and output validity are available in [17].

Ken's most interesting property is *composable reliability*. Consider two systems of independently developed Ken processes. The two systems separately and individually enjoy the strong global correctness guarantees of distributed consistency and output validity. If they begin exchanging messages with one another, then the global correctness guarantees immediately expand to cover the *union* of the systems. The developers of the two systems took no measures whatsoever to make this happen. In particular they did not need to anticipate inter-operation between the two systems. Ken's reliability measures require no coordination among processes for checkpointing during failure-free operation, for recovery, or for output.

Ken furthermore brings important "social" benefits to software development. Because its reliability measures are purely local and independent, Ken *contains* damage rather than propagating it and *focuses* responsibility rather than diffusing it. For example, a crash of one Ken process does not trigger rollbacks of any other process; a remarkable number of prior rollback-recovery schemes do *not* have this property. The net effect is that Ken is unlikely to cause finger-pointing among teams responsible for designing and operating different components.

Finally, Ken is implementation-friendly in several ways. It is frugal with durable storage. Because recovery requires only the most recent local checkpoint, older checkpoints may be deleted. Ken never takes useless checkpoints [17]. Checkpoints are small as they include only the persistent heap, not the stack or kernel state; whole-process checkpoints taken at the OS or virtual machine monitor layer would be larger. An implementation may take checkpoints incrementally, as ours does. It is furthermore possible to delta-encode and/or compress checkpoints, though our current implementation does neither. Finally, Ken admits implementation as a lightweight, compact, portable library. Our stand-alone Ken implementation is available as open source software [16].

4 Event-Driven State Machine Integration

In this section, we describe integration of the Ken reliability protocol with an event-driven state machine toolkit. The concepts of common event-driven programming paradigms and Ken are complementary, allowing a seamless integration nearly transparent to developers.

4.1 Design

Event-driven programming has long been used to develop distributed systems because it matches the asynchronous environment of a networked system. In event-driven programming, a distributed system is a collection of event handlers reacting to incoming events. For example, events may be network events like message delivery, or timer events like a peer response timeout. To prevent inconsistency and avoid deadlock, event-driven systems frequently execute atomically, allowing a single event handler at a time. Event handlers are non-blocking, so programmers use asynchronous I/O, continuing execution as needed through dispatching subsequent events.

All execution therefore takes place during event handlers, and importantly, all outputs are generated therein. Typically, a single input is fed to the event handler, and it must run to completion without further inputs. To conform to the event loop, most event-driven toolkits contain specific I/O libraries for messaging—one that provides

```
while (running) {
  readyEvents = waitForEvents(sockets, timers);
  for (Event e in readyEvents) {
    if (Ken.isDupEvent(e)) { continue; }
    Ken.blockOutput();
    dispatchEvent(e); //becomes ken_handler()
    Ken.writeEOTFile();
    Ken.transmitAckAndOutputs(e);
} }
```

Figure 3: Common event loop with Ken integration

message I/O results only in subsequent events (i.e. fire-and-forget messaging).

Figure 3 shows a typical event loop for a distributed system. A single thread waits for network and timer events, then dispatches all ready events by calling their event handlers in turn. To integrate Ken, we need only verify that the input is new, block outputs by buffering them in the event library, and then acknowledge the input and externalize the outputs once the end-of-turn file is written to non-volatile storage. As there is only one thread, the EOT file will be consistent with the turn state. Finally, we replace the dispatch function with the Ken handler function, to provide access into the persistent heap.

4.2 Implementation

We now describe the integration of the Ken protocol with Mace [20], an open-source, publicly available distributed systems toolkit. To fully integrate Ken into Mace, we replaced the networking libraries with Ken persistent message handling and acknowledgments, replaced the facility for scheduling application timers with a Ken callback mechanism, replaced memory allocation in Mace with `ken_alloc()`, and connected the Ken `handler()` function to the Mace event processing code. Finally, we relinquished control over application startup to Ken.

Mace and Ken appeared to be a perfect fit for each other, as Mace provided non-blocking atomic event handlers, explicit persistent state definition, and fire-and-forget messaging. However, in the implementation integrating Mace and Ken we ran across numerous complicating details. Thankfully, these are largely transparent to the *users* of MaceKen, and need only be implemented in the MaceKen runtime. We now discuss a few of these.

State checkpoints. Mace provides explicit state definition, so we intended to checkpoint the explicit state only. However, many of the variables were collections based on the C++ Standard Template Library (STL), which internally handles memory management. This complicates checkpointing as the STL collections contained references to many dynamic objects. Instead, we replaced the global allocator, requiring all Mace heap variables to be maintained by Ken, even transient and temporary state. This exercise also caused us to streamline some runtime libraries to reduce the number of pages unnecessarily dirtied.

Initialization. Unlike Mace, Ken requires that the implementation of `main()` be defined within Ken, and not by a user program. This gives Ken control over application initialization, to set up Ken state appropriately without application interference, and also hiding application restart. As Mace allowed substantial developer flexibility on application initialization, this created some tension. We had to incorporate a MaceKen-specific initialization function that MaceKen would call on each start, to properly initialize certain state; however, this is hidden from users to preserve the MaceKen illusion that a program never fails. Ultimately, it makes both Mace and MaceKen easier to use—developers need not worry about complex system initialization.

Event Handlers. Mace provides atomic event handling by using a mutex to prevent multiple events from executing simultaneously. This design allows multiple threads to attempt event delivery, such as one set of threads delivering network messages, and another set of threads delivering timer expirations. This design is at odds with other common event dispatch designs where all event processing is done through a common event loop executed by a single thread. Ken assumes such a common, monolithic event loop, which required adding an event-type (e.g. message delivery, timer expiration, etc.) dispatch layer prior to the event handler dispatch Mace already used, adding additional overhead.

Transport Variants. By default Ken re-transmits messages until acknowledged, doubling the timeout interval with each re-transmission (i.e., exponential backoff). This strategy is based on the principle that in our target environments, network losses are infrequent and most retransmissions will be due to restarting Ken processes. However for communication-intensive applications, e.g., our graph analysis (Section 5.2), the volume and rate of communication increase the chances of loss due to limited buffer space in the network or hosts. To prevent excess retransmissions, we implemented two additional transport variants in addition to the original Ken default: First, we added Go-Back-N flow control atop the UDP-based protocol to minimize latency for message-intensive applications in situations where receive buffers are likely to fill. We have also implemented a TCP-based transport that simply re-transmits in response to broken sockets. The distributed graph analysis experiment of Section 5.2 and the distributed storage tests of Section 5.3 employ the TCP transport; the microbenchmarks of Section 5.1 used the default Ken mechanism.

Logging. Mace contains a sophisticated logging library that is not suitable for unbuffered `stderr` output.

As a result, we had to rewrite the library to specially use standard heap objects, in many cases replacing provided containers whose allocation we could not control. As with `stderr`, logging must be used as a write-only mechanism or MaceKen warranties are voided.

Our MaceKen implementation will be released as open source software [18].

5 Evaluation

We tested both our stand-alone Ken implementation and also MaceKen to verify that they deliver Ken's strong fault tolerance guarantees, to measure performance, and to evaluate usability.

5.1 Microbenchmarks

We conducted microbenchmark tests to measure Ken performance (turn latency and throughput) on current hardware and to estimate performance on emerging non-volatile memory (NVRAM). One or more pairs of Ken processes on the same machine pass a zero-initialized counter back and forth, incrementing it by one in each turn, until it reaches a threshold. The rationale for using two Ken processes rather than one is that our test scenario involves two reliability guarantees, local state reliability and reliable pairwise-FIFO messaging, whereas incrementing a counter once per turn in a single Ken process would not involve any of Ken's message layer. We ran our tests on a 16-core server-class machine with 2.4 GHz Xeon processors, 32 GB RAM, and a mirrored RAID storage system containing two 72 GB 15K RPM enterprise-class disks; the RAID controller contained 256 MB of battery-backed write cache. The storage system is configured to deliver enterprise-grade data durability, i.e., all-important foundations such as `fsync()` and `fdatasync()` work as advertised (our tests employ the latter, which is sufficient for Ken's guarantees).

We tested Ken in three configurations: the default mode in which `fdatasync()` calls guarantee checkpoint durability at the end of every turn; "no-sync" mode, in which we simply disable end-of-turn synchronization; and "`tmpfs`" mode, in which Ken commits checkpoints to a file system backed by ordinary volatile main memory rather than our disk-backed RAID array. The no-sync case allows us to measure performance for weakened fault tolerance—protection against process crashes only and not, e.g., power interruptions or kernel panics. The `tmpfs` tests shed light on what performance might be on future NVRAM.

We measure light-load latency by running only two Ken processes that pass a counter back and forth, incrementing it to a final value of 15,000 (150,000 for the `tmpfs` scenario). Default reliable Ken with `fdatasync()` averages 4.27 milliseconds per turn. Recall from Section 3.2 that Ken synchronizes twice at the end of each turn, once for the end-of-turn (checkpoint) file and once for the parent directory. The expected time for each call should roughly equal a half-rotation latency, which on our 15K RPM disks is 2 ms. Without end-of-turn synchronization, Ken's turns average 0.575 ms, roughly 7.4× faster; `tmpfs` further reduces turn latency to 0.468 ms.

The throughput of a single pair of Ken processes in our "counter ping-pong" test is limited by turn latency because the two Ken processes' turns must strictly alternate. To measure the aggregate turn throughput capacity of our *machine*, we vary the number of Ken processes running. As in our latency test, pairs of Ken processes increment a counter as it passes back and forth between them. With data synchronization on our RAID array, throughput increases gradually to a plateau, eventually peaking at over 1,750 turns per second when several hundred Ken processes are running. Without end-of-turn data synchronization, throughput peaks at over 6,000 turns per second when roughly sixteen Ken processes are running. On `tmpfs`, peak throughput exceeds 20,700 turns per second with 22 Ken processes running.

As expected, Ken's performance depends on the underlying storage technology and its configuration. Our enterprise-grade RAID array provides reasonable performance for a disk-based system. Our no-sync measurements show that latency drops substantially and throughput increases more than 3× if we relax our fault tolerance requirements. NVRAM would provide the best of both worlds: even lower latency and an additional 3× throughput increase over the no-sync case, without compromising fault tolerance. We have not yet conducted tests on flash-based storage devices but we expect that SSDs will offer substantially better latency and throughput compared to disk-based storage. At the other end of the spectrum, on a system with simple conventional disk storage, we have measured Ken turn latencies as slow as 26.8 ms.

For applications that must preserve data integrity in the face of failures, the important question is whether general-purpose integrity mechanisms such as Ken make efficient use of whatever physical storage media lie beneath them. In tests not reported in detail here, we found that Ken's transactional turns are roughly as fast as ACID transactions on two popular relational databases, MySQL and Berkeley DB. On a machine similar to the one used in our experiments, ACID transactions take a few milliseconds for both Ken as well as the RDBMSes. This isn't surprising because the underlying data synchronization primitives provided by general-purpose operating systems are the same in these three systems, and the underlying primitives dominate light-

load latencies. Our finding merely suggests that gratuitous inefficiencies are absent from all three. Ken offers different features and ergonomic tradeoffs compared to relational databases—it provides reliable communications and strong global distributed correctness guarantees, but not relational algebra or schema enforcement, for example—and comprehensive fair comparisons are beyond the scope of the present paper.

5.2 Transparent Checkpoints: Distributed Graph Analysis

Recent work has applied Mace to scientific computing problems far removed from the systems for which Mace was originally intended [36]. In a similar vein, we further tested MaceKen's versatility by employing it for a graph analysis problem used as a high-performance computing benchmark [14]: The maximal independent set (MIS) problem. Given an undirected graph, we must find a subset of vertices that is both independent (no two vertices joined by an edge) and maximal (no vertex may be added without violating independence). A graph may have several MISes of varying size; the problem is simply to find any one of them.

We implemented a distributed MIS algorithm [31] that lends itself to MaceKen's event-driven style of programming. Like many distributed MIS solvers, this algorithm has a high ratio of communication to computation and so distribution carries a substantial inherent performance penalty. Our fault-tolerant distributed solver is actually *slower* than a lean non-fault-tolerant single-machine MIS solver when applied to random Erdős-Rényi graphs that fit into main memory on a single machine. However distribution is the only way to tackle graphs too large for a single machine's memory, and our MaceKen solver can exploit an entire cluster's memory. More importantly, our MaceKen solver can survive crashes, which is important for long-running scientific jobs.

To test MaceKen's resilience in the face of highly correlated failures, we ran our MIS solver on a graph with 8.3 million vertices and 1 billion edges on 20 machines and then simultaneously killed all MaceKen processes during the job. When we re-started the processes, the distributed computation resumed where it left off and completed successfully. We carefully verified that its output was identical to that of a known-correct reference implementation of the same algorithm. Our experiences strengthen our belief that MaceKen can be appropriate for scientific computing, and furthermore demonstrate that Ken can transparently add fault tolerance with zero programmer effort.

The largest single graph that our MaceKen MIS solver has tackled has 67 million vertices and 137 billion edges;

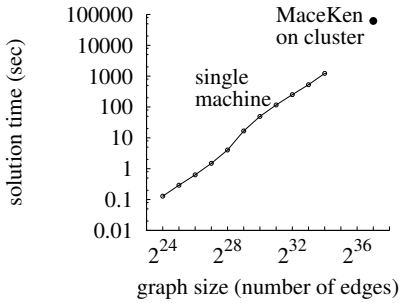


Figure 4: MIS: single machine vs. MaceKen cluster

a straightforward representation of this graph requires over 1.1 TB. Running on a 200-machine cluster, our MaceKen MIS solver took 8.96 hours to generate this random graph and 17.07 hours to compute an MIS with end-of-turn data synchronization disabled. Figure 4 compares this run time with the run times of a lean and efficient single-machine MIS solver on smaller graphs; all graphs are Erdős-Rényi graphs and the number of edges is 2048× the number of vertices. The single-machine solver is not based on Ken or MaceKen and it is not fault tolerant. Our distributed MaceKen solver can tackle graphs 8× larger than our single-machine solver. Figure 4 suggests that, given enough memory, the single-machine solver would probably run faster, but we do not have access to a single machine with 1.1 TB of RAM. Our results suggest that a MaceKen graph analysis running on sufficiently fast durable storage (NVRAM) can provide both fault tolerance and reasonable performance.

5.3 Survivability: Distributed Storage

We conducted experiments on a Mace implementation of the Bamboo Distributed Hash Table (DHT) protocol [32] in the face of churn. No modifications were made to Bamboo to enable the Ken protocol—the Mace implementation compiled and ran directly with MaceKen. We chose to work with the Bamboo protocol because it was specifically designed to tolerate and survive network churn (short peer node lifespan). Bamboo uses a rapid join protocol and periodic recovery protocols to correct routing errors and optimize routing distance. Our own work [20, 19] has confirmed the results of others that Bamboo delivers consistent routing under aggressive churn. However, this work has focused on consistent *routing*, not the preservation of DHT data maintained atop Bamboo routing, which was expected to pose additional challenges and not be durable. The DHT is a separate implementation that uses Bamboo for routing, but is responsible for storage, replication, and recovery from failure. As the consistency and durability of data stored at failed DHT nodes on peer computers are suspect at

best, traditional designs use in-memory storage only, and tolerate failures through replication instead. If a node reboots, it will rejoin the DHT and reacquire its data from peer replicas. Mace includes such a basic DHT, which we used for testing.

In exploring the Bamboo protocol's resilience to churn, we initially discovered that even under periods of relatively high churn (mean lifetime of 20 seconds), the Bamboo DHT is able to recover quickly and maintain the copies of stored data. While pleasantly surprised, we determined that this occurred as a direct result of the fast failure notification that surviving peers receive when a remote DHT process terminates and sockets are cleaned up by operating systems. However, in the case of power interruptions, kernel panics, or hardware resets, the socket state is *not* cleaned up but rather is erased with no notice at all. TCP further will not time-out the connection for several minutes after the surviving endpoint attempts to send data, delaying failure recovery. Once the physical machine has resumed operation, the OS will respond to old-socket-packets with a socket reset, causing the failure to be detected sooner. However, in both cases, failure is not detected unless the surviving endpoint attempts to send new data. As obtaining access to a large cluster where we can control the power cycling of machines is impractical, we sought to devise an alternate mechanism to conduct data survivability and durability tests.

Linux Containers (LXCs) [26] are a lightweight virtualization approach based on the concepts of BSD jails. Importantly, network interfaces can be bound within an LXC, with their own network stack of which the host operating system is unaware. We configured our experiment to use LXCs for running DHT nodes. For each LXC, a virtual Ethernet device is created, with one endpoint inside the LXC and the other in the host OS. The host OS then routes packets from the LXC to the physical network over the real Ethernet device. When we wished to fail a DHT node, we could first remove the host's virtual Ethernet device endpoint to prevent the network stack in the LXC from sending any packets. While killing the processes next caused attempts to cleanup the socket state, these failed due to lack of connectivity. Finally, destroying the LXC destroyed all evidence of the socket, allowing the LXC to be restarted without having TCP attempt to resend the socket FIN.

We conducted experiments to mimic failure scenarios likely to be observed in managed infrastructures. We ran 300 DHT nodes on 12 physical machines, using ModelNet [34] to emulate a low-latency topology with three network devices, and 100 DHT nodes connected to each. In our experiments, after an initial stabilization period, DHT clients would periodically put new data in the DHT, and request data, split between just-added data (Get) and previously-added data (Prior). Get requests commence ten minutes after start. Prior requests commence after 45 min to ensure that the DHT contains sufficient data. Our experimental setup places many DHT nodes on each physical machine, and if DHT nodes called `fsync()` the machine's storage system would be overloaded. We therefore emulate the latency of `fsync()` calls by adding 26 ms sleep delays. The slowest Ken *per-turn* latencies that we have observed are roughly 26 ms; since a Ken turn involves *two* `fsync()` calls, by adding 26 ms to each `fsync()` call our experiments measure Ken performance pessimistically/conservatively.

Since a DHT uses replication to increase availability and data survivability upon crashes, we have configured the unmodified Mace DHT implementation to have five replicas including the primary store. With Ken enabled, no replication is needed to survive crash-restart failures, so the MaceKen DHT stores data only on the primary store in these experiments.

In the middle of the experiment we tested two kinds of failures: first a "power interruption" that restarted all DHT nodes except a distinguished bootstrap node, and second a "rolling restart" that restarted each node twice over a period of 5 minutes, such as for urgent, unplanned maintenance to the entire cluster. Each restarted node is offline for only 5 seconds before being restarted—chosen to maximize the unmodified (i.e., non-Ken) Mace DHT implementation's ability to recover quickly—real operating systems currently take considerably longer to restart.

Figures 5 and 6 present success fractions of types of DHT lookups. For both experiments, when using the MaceKen runtime, no impact can be seen in the correct operation of the DHT storage, either for Get or Prior. When the unmodified Mace runtime is used, failures cause a period of disruption to DHT requests. When nodes failed simultaneously, after the period of disruption the Get requests resume delivering fast, successful responses, but most Prior requests fail due to permanent data loss when all replicas of the data simultaneously failed. In the rolling-restart case, some data survived because the DHT could detect failure of some replicas while other replicas still survived and could further replicate the data. If all machines failed before any of them detected failures and could replicate, then the data were lost.

Figures 7 and 8 show average latency for all the requests in each minute. MaceKen overheads roughly double the cost of the DHT lookups compared to Mace, but this is reasonable performance, particularly given the success fractions, and the slow storage device being simulated. During and immediately after the failures, MaceKen performance is slower because it is performing recovery, patching, and reliable data retransmission.

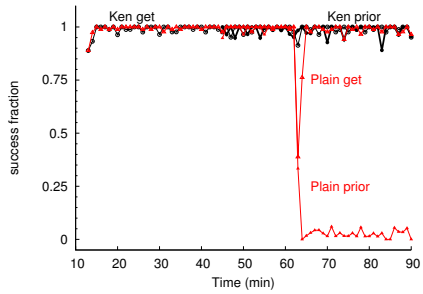


Figure 5: Simultaneous Failure

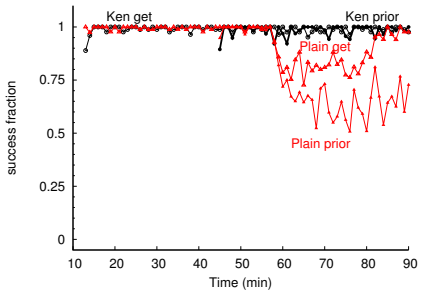


Figure 6: Rolling-restart

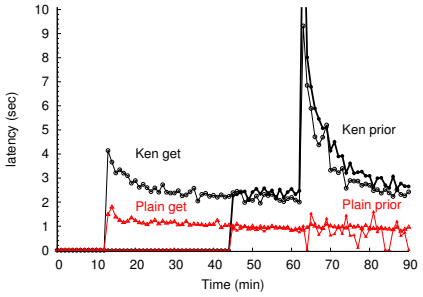


Figure 7: Simultaneous Failure

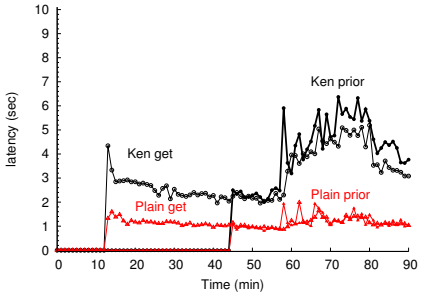


Figure 8: Rolling-restart

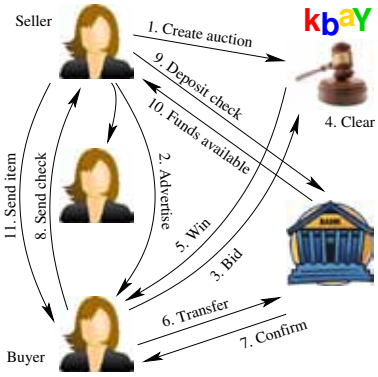


Figure 9: "kBay" e-commerce

As we made no modifications to Bamboo to use MaceKen, and based on Bamboo survivability in these experiments, we conclude that the MaceKen runtime is both easy to use and increases the survivability of existing peer-to-peer systems.

5.4 Composable Reliability: E-Commerce

Decentralized software development is the rule rather than the exception for complex distributed computing systems. To take a familiar example, "mashups" compose independently developed and managed Internet services in client Web browsers [35]. Other important examples, e.g., supply-chain management software, lack a single point of composition but nonetheless require end-to-end reliability across software components designed and deployed by teams separated by time, geography, and organizational boundaries. Ken is well suited to such systems because it guarantees reliability that composes globally despite being implemented locally.

Our "kBay" e-commerce scenario (Figure 9) stress-tests Ken's composable reliability. Sellers create auctions and advertise items for sale among friends, who bid on items. The kBay server clears auctions and notifies winners. Winning bidders must transfer money from savings to checking accounts before sending checks to sellers. Sellers deposit checks, causing a transfer from the buyer's checking account to the seller's savings account. If the check does not "bounce" the seller sends the purchased item to the buyer. Without Ken, crashes and message losses could create several kinds of mischief for kBay, e.g., causing the bank to destroy money or create counterfeit, causing the auction site to prematurely remove unsold items from the marketplace or award the same item to multiple buyers, causing checks to bounce or causing check writers to forget having written them. Similar problems have long plagued real banks [2] and e-commerce sites [3, 33].

Given complete control over all kBay software, a single careful development team could in principle guarantee global distributed consistency and output validity. Ken's composable reliability makes it easy for *separate* teams to implement components independently and still achieve global reliability without coordination. We implemented atop Ken all of the components depicted in Figure 9. In our tests, 32 clients offer items for sale via the auction server and advertise them among five other clients. Injected failures repeatedly crash the auction server, the bank, and the clients, which ran on three separate machines. We verified output validity by checking that every item is eventually sold to exactly one buyer and that the sum of money in the system is conserved. Our results confirm our expectations: Ken guarantees global correctness even in the presence of repeated crash/restart failures of stateful components designed without coordinated reliability.

6 Related Work

Previous work related to our efforts falls under three main umbrellas. First, there is currently a popular set of systems for managing cluster computation. These spe-

cial case systems, while effective for their goals, are not general enough to support the range of applications we are targeting. Second, a host of toolkits for building varieties of distributed systems exist. However, these have typically targeted developing wide-area peer-to-peer systems. They do not provide the proposed combination of data center optimization, performance tuning, and reliability. Finally, the Ken design follows on a line of rollback-recovery research, applied to general purpose systems. Our proposed work shows how to apply the advances Ken makes in this line in a generic way to the development of a broad class of applications.

Cluster Computation Systems Cluster computing infrastructures include two broad classes. First, job scheduling engines such as Condor [22, 11] are designed to support batch processing for distributed clusters of computers. These schedulers tend to be focused on efficient scheduling of a large set of small-scale tasks, such as for a single machine, across a wide set of resources. More recently, systems such as MapReduce [8], Ciel [28] and Dryad [15], have emerged, and focus on how to partition single, large-scale data-parallel computations across a cluster of machines. Both classes support process failures, but the implementation is predominantly focused on batch processing. In batch processing, failure handling is much simpler, because only the eventual result is emitted as final output. Failures can therefore be tolerated by simply re-computing the result, possibly using cached partial earlier results.

MaceKen is suitable for developing non-batch applications that emit results continuously and for applications with continuous inputs and outputs. In non-batch applications, simply restarting crashed processes does not guarantee distributed correctness. In particular, we focus our design on approaches yielding distributed consistency and output validity, where the output remains acceptable despite tolerated failures. Consider, for example, the CeNSE application [23]. CeNSE includes a large group of sensors and actuators embedded in the environment, connected with an array of networks. These actuators provide near-real-time outputs, so the application's job is not to perform batch computations, but rather to generate outputs continuously. In contexts like CeNSE, output validity is critical.

Programming Distributed Systems There are many toolkits for building distributed systems. We built our system on top of the Mace toolkit, which includes a language, runtime, model checker, simulator, and various other tools [20]. But Mace, like many other toolkits, is focused on the class of general, wide-area distributed systems such as peer-to-peer overlays.

Other similar toolkits include P2 [24], Libasync and Flux [27, 37, 4], and Splay [21]. P2 utilizes the Overlog declarative language as an efficient way to specify overlay networks. Its data-flow design lends to effective parallelization, but its performance is not optimized for data centers. Libasync, its parallelization companion, libasync-mp, and event language Flux, provide another highly optimized toolkit for running event-driven and parallel programs. But again, the focus is on distributed event processing with asynchrony, not its combination or ability to handle automated rollback-recovery. Splay's focus, beyond the basic language and runtime, focuses on deployment and fair resource sharing across applications, and does not target data center environments.

To target data centers, MaceKen focuses on adding reliability to common data center failure-restart conditions which are not the expected failure case in wide-area distributed systems. Additionally, MaceKen targets resource usage based on expected resource availability in emerging data centers—network bandwidth is assumed to be abundant, and non-volatile RAM is available for efficient local checkpointing, while still needing to be frugal with system memory.

Rollback Recovery Rollback-recovery protocols have a long history [9]. These protocols include both checkpoint-based and log-based protocols, which differ in whether they record the state of a system or log its inputs. Combinations of checkpoint- and log-based systems continue to be popular, such as Friday [13], which uses system logs on a distributed application to run a kind of gdb-like debugger. A major challenge in rollback-recovery systems is to be able to rollback or replay state efficiently across an asynchronously connected set of nodes; in addition to checkpointing overhead, *coordination* overhead can be significant both in failure-free operation and during recovery. Moreover, systems that prevent failures from altering distributed computations in any way at all are overkill for a broad class of distributed systems that require only the weaker guarantee of output validity. Accepting this weaker guarantee actually provides Ken two distinct advantages: first, output validity can be preserved with a simple coordination-free local protocol, and second, it can actually allow a system to survive in some cases when the original sequence of events would lead to a persistent failure. See Lowell et al. for a detailed discussion of how and to what extent nondeterminism helps systems like Ken to recover from failures [25]. Ken differs from the well-known folklore approach of checkpointing atomically with every message/output because Ken bundles messages and outputs into turns, which simplifies the implementation and provides transactional turns that facilitate reasoning about distributed event-driven computations. Correia et al. execute event handlers on multiple local state copies to detect and contain arbitrary state corruption [7]. This complements Ken's crash-resilience handling by automating corruption checks.

7 Conclusions

The Ken rollback-recovery protocol protects local process state from crash/restart failures, ensures pairwise-FIFO message delivery and exactly-once message processing, and provides strong global correctness guarantees for distributed computations—distributed consistency and output validity. Ken's reliability guarantees furthermore *compose* when independently developed distributed systems interact or merge. Ken complements high-level distributed systems toolkits such as Mace, which raise the level of abstraction on asynchronous event-driven programming. Our integration of Ken into Mace simplifies Mace programs and enables them to adapt to new managed environments prone to correlated failures. Our tests show that our integrated MaceKen toolkit is versatile enough to tackle distributed programming problems ranging from graph analyses to DHTs, providing crash resilience across the board.

Acknowledgments

We would like to thank our shepherd, John Regehr, and our anonymous reviewers for their helpful comments. Research support is provided in part through the HP Labs Open Innovation Research Program. This research was partially supported by the Department of Energy under Award Number DE–SC0005026. See http://www.hpl.hp.com/DoE-Disclaimer.html for additional information.

References

[1] L. Alvisi, E. Elnozahy, S. Rao, S. A. Husain, and A. D. Mel. An analysis of communication induced checkpointing. In *Fault-Tolerant Computing*, 1999. doi:10.1109/FTCS.1999.781058.

[2] R. J. Anderson. Why cryptosystems fail. *Commun. ACM*, 37, Nov. 1994. doi:10.1145/188280.188291.

[3] S. Ard and T. Clark. eBay blacks out yet again, June 1999. http://news.cnet.com/eBay-blacks-out-yet-again/2100-1017_3-226987.html.

[4] B. Burns, K. Grimaldi, A. Kostadinov, E. D. Berger, and M. D. Corner. Flux: A language for programming high-performance servers. In *USENIX ATC*, 2006.

[5] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of a distributed system. *ACM TOCS*, 3(1):63–75, Feb. 1985. doi:10.1145/214451.214456.

[6] T. Close. Waterken, 2009. http://waterken.org/.

[7] M. Correia, D. Ferro, F. P. Junqueira, and M. Serafini. Practical hardening of crash-tolerant systems. In *USENIX ATC*, 2012.

[8] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, 2004. acmid:1251264.

[9] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34:375–408, Sept. 2002. doi:10.1145/568522.568525.

[10] M. Finifter, A. Mettler, N. Sastry, and D. Wagner. Verifiable functional purity in Java. In *ACM CCS*, 2008. acmid:1455793.

[11] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[12] V. K. Garg. *Elements of Distributed Computing*. Wiley, 2002.

[13] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: Global comprehension for distributed replay. In *NSDI*, 2007. http://www.usenix.org/event/nsdi07/tech/geels.html.

[14] The Graph500 Benchmark. http://www.graph500.org/.

[15] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS OS Rev.*, 41:59–72, Mar. 2007. acmid:1273005.

[16] T. Kelly. http://ai.eecs.umich.edu/~tpkelly/Ken/.

[17] T. Kelly, A. H. Karp, M. Stiegler, T. Close, and H. K. Cho. Output-valid rollback-recovery. Technical report, HP Labs, 2010. http://www.hpl.hp.com/techreports/2010/HPL-2010-155.pdf.

[18] C. Killian. http://www.macesystems.org/maceken/.

[19] C. Killian, K. Nagaraj, S. Pervez, R. Braud, J. W. Anderson, and R. Jhala. Finding latent performance bugs in systems implementations. In *FSE*, 2010. doi:10.1145/1882291.1882297.

[20] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: language support for building distributed systems. In *PLDI*, 2007. doi:10.1145/1250734.1250755.

[21] L. Leonini, É. Rivière, and P. Felber. Splay: Distributed systems evaluation made simple. In *NSDI*, 2009. Available from: http://www.usenix.org/event/nsdi09/tech/.

[22] M. Litzkow, M. Livny, and M. Mutka. Condor-a hunter of idle workstations. In *ICDCS*, volume 43, 1988.

[23] S. Lohr. Smart dust? Not quite, but we're getting there. *New York Times*, Jan. 2010. http://www.nytimes.com/2010/01/31/business/31unboxed.html.

[24] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *SOSP*, 2005. doi:10.1145/1095810.1095818.

[25] D. E. Lowell, S. Chandra, and P. M. Chen. Exploring failure transparency and the limits of generic recovery. In *OSDI*, 2000.

[26] lxc Linux containers. http://lxc.sourceforge.net/.

[27] D. Mazières. A toolkit for user-level file systems. In *USENIX ATC*, 2001.

[28] 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*, 2011. http://www.usenix.org/event/nsdi11/tech/full_papers/Murray.pdf.

[29] T. Negrino and D. Smith. *JavaScript and AJAX*. Peachpit Press, seventh edition, 2009.

[30] http://linux-mm.org/OverCommitAccounting.

[31] D. Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM Press, 2000.

[32] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *USENIX ATC*, 2004. http://www.usenix.org/event/usenix04/tech/general/rhea.html.

[33] I. Steiner. eBay blames search outage on listings surge, Nov. 2009. http://www.auctionbytes.com/cab/abn/y09/m11/i21/s02.

[34] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *OSDI*, 2002. http://www.usenix.org/event/osdi02/tech/vahdat.html.

[35] H. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in MashupOS. In *SOSP*, 2007.

[36] S. Yoo, H. Lee, C. Killian, and M. Kulkarni. Incontext: simple parallelism for distributed applications. In *HPDC*, 2011. doi:10.1145/1996130.1996144.

[37] N. Zeldovich, A. Yip, F. Dabek, R. Morris, D. Mazières, and F. Kaashoek. Multiprocessor support for event-driven programs. In *USENIX ATC*, June 2003.

Managing Large Graphs on Multi-Cores With Graph Awareness

Vijayan Prabhakaran, Ming Wu, Xuetian Weng
Frank McSherry, Lidong Zhou, Maya Haridasan[†*]
Microsoft Research, [†]Google

Abstract

Grace is a graph-aware, in-memory, transactional graph management system, specifically built for real-time queries and fast iterative computations. It is designed to run on large multi-cores, taking advantage of the inherent parallelism to improve its performance. Grace contains a number of graph-specific and multi-core-specific optimizations including graph partitioning, careful in-memory vertex ordering, updates batching, and load-balancing. It supports queries, searches, iterative computations, and transactional updates. Grace scales to large graphs (e.g., a Hotmail graph with 320 million vertices) and performs up to two orders of magnitude faster than commercial key-value stores and graph databases.

1 Introduction

Last decade has witnessed an increase in the number and relevance of graph-based applications. Finding shortest path over road networks, computing PageRank on web graphs, and processing updates in social networks are a few well-known examples of such real-world workloads, which access graphs with millions and billions of vertices and edges.

Some of these workloads – such as PageRank [10] – run in the background, without any direct user interactions. Since they do not have latency constraints, they can be run on existing data-parallel architectures such as MapReduce [12], Hadoop [4], or DryadLinq [32]. Given the importance of these workloads, new distributed architectures such as Pregel [21] are built to run them more efficiently. Typically, such batch-processed workloads run on read-only graph snapshots and therefore, their platforms do not support graph updates.

On the other hand, certain graph workloads are latency sensitive. For example, finding directions in a map or a search query accessing the social network of a user to find relevant information require responses that do not exceed strict time constraints (typically, in the order of few 10s of milliseconds), even though each such query (or workloads) can access millions of random vertices and edges.

Whereas batched workloads run on static graph snapshots, real-time queries often run on graphs that can change continuously (such as in social network). Existing graph processing platforms are largely unsuitable to host these workloads because they are optimized for batched, read-only workloads. Other general purpose systems such as key-value stores or relational databases are graph-agnostic and offer sub-optimal performance.

Grace is a graph-aware, in-memory, transactional graph management system that is specifically designed for supporting low-latency graph applications. It exposes a simple graph querying interface and a platform for running iterative graph computations. Grace supports transactions, guaranteeing ACID semantics on graph modifications. Consistent graph snapshots can be created instantaneously in Grace, allowing read-only workloads – such as computing the shortest path – to run concurrently with other transactional updates.

Two design aspects of Grace distinguish it from existing systems, such as key-value stores and relational databases that are used for storing graphs. First, Grace is *graph-aware*, that is, Grace's design including its data structures, algorithms, policies, and interfaces are chosen to support graphs and graph workloads rather than being general purpose. Second, given that many graph workloads are highly parallelizable, Grace is optimized to run on large-scale multi-cores, taking advantage of the underlying parallelism and memory layout to improve the overall performance.

We show that this graph-awareness and support for parallelism can boost performance significantly when compared to key-value stores such as Berkeley DB (BDB) [1] and even when compared to other commercially available graph databases such as Neo4j [6]. Grace runs up to two orders of magnitude faster than BDB and Neo4j under a single thread; under multi-threaded mode, Grace runs up to 40 times faster than its unoptimized, single threaded mode. Grace scales well on large multi-cores and large graphs; for example, it runs on a 48-core machine, managing a 320 million node Hotmail graph.

In the rest of the paper, we present the system as fol-

[*]Work done at Microsoft Research

lows. Next, we present the high-level design of Grace. In Section 3, we describe the algorithms used for graph partitioning and ordering vertices in memory. Following that, we explain the platform for running iterative computations in Section 4 and present the details of transactions in Section 5. We then explain the implementation details in Section 6 and present results from evaluation in Section 7. Finally, we discuss related work in Section 8 and conclude in Section 9.

2 Design for a Graph Management System

Grace's overall design is driven by the following two characteristics exhibited by several graph workloads.

2.1 Imparting Graph Awareness

A graph's inherent structure – which reflects how its vertices are connected – gives us certain hints about how the vertices will be accessed. Most graph workloads follow a strong *graph-specific locality* in their access patterns; specifically, when a workload accesses a vertex, it is highly likely to access the vertex's neighbors in the near future. For example, in a social network, sending updates to friends (and friends of friends) involves traversing the social graph and updating neighbors' data. Even iterative computation such as PageRank, which does not follow any graph-traversal pattern, computes the rank of a page and passes a fraction of its rank to its neighbors (thereby, accessing its neighbors' data).

However, general-purpose solutions – such as key-value stores and relational databases – used today for storing and accessing graph datasets are often agnostic to the underlying graph structure and therefore rely only on default access patterns and policies to improve performance.

Grace's graph-awareness spans the whole system starting from its low-level cache and memory layouts to high-level interfaces exposed to applications. Some of its graph-aware features include a fast graph partitioning algorithm, which can be used to split a graph into smaller sub-graphs and decide where to assign new graph updates, a graph-aware layout where vertices are ordered such that neighboring vertices are placed close to each other, and a set of querying and updating APIs for accessing a graph.

2.2 Embracing Parallelism

Second, graph workloads are often partitionable and therefore, parallelizable. For example, in a social network, two different users can update their status in parallel without affecting each other. Even computations such as PageRank can be run in parallel on a partitioned graph such that updates from a partition are sent to another partition only if they share an edge. This partitionable nature of graphs and their workloads is well-aided by the evolution of hardware into the multi-core era.

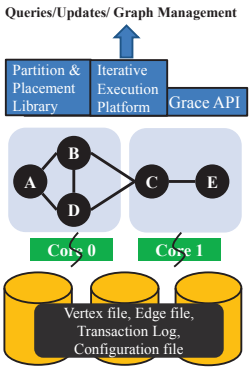


Figure 1: **Grace Architecture.**

However, general purpose systems often lack support for large-scale multi-cores; for example, BDB and Neo4j do not support partitioned data or computation. Even if they are built for multi-cores, because of the absence of graph-awareness, key-value stores and relational databases cannot efficiently partition or layout a graph on multi-cores.

Grace makes several optimizations for multi-core processors. For example, it minimizes inter-core communications by batching updates across partitions. Grace's load-balancer can dynamically shed load from a thread that is handling a 'hot' graph partition to other free threads.

An overall result of these optimizations is that Grace outperforms other comparative systems; in a single-threaded mode, Grace runs up to two orders of magnitude faster than Berkeley DB and Neo4j; in a multi-threaded mode, Grace gains a speed up of up to 40 times relative to its single-threaded performance.

2.3 Grace Architecture

A high-level sketch of Grace is presented in Figure 1. Grace runs on a single machine. Since low-latency of queries is one of our main objectives, we specifically choose to keep the entire graph in-memory. Grace accesses the durable media only when loading a graph initially and logging transactional updates.

Grace exposes a set of APIs for querying and updating a graph. Queries can be vertex-specific (*e.g.*, GetVertexAndNeighbors()), or partition or graph-specific (*e.g.*, GetNumberOfVertices()). A graph can be updated by adding or deleting vertices and edges.

Grace also presents a platform for running iterative computations – which in turn use the querying APIs – on graph partitions. While iterative computations (such as PageRank) are typically treated as batch-processed workloads, we show that when run on Grace, they take a few minutes instead of several hours to complete.

Within Grace, a graph can be partitioned and stored as smaller sub-graphs. Grace provides a library for running management tasks such as partitioning a graph or rearranging its vertices in-memory; parameters for partitioning such as the algorithm to use and number of partitions to create can be specified via a configuration file. Each partition is handled by a thread, which can be pinned to a core; the thread is responsible for processing queries and updates to vertices and edges of that partition.

Next, we present more details on how Grace partitions a graph and arranges its vertices in memory.

3 Partitioning and Placement

One of the key opportunities we have in designing a specialized store for graph structured data is to exploit the expected locality of reference in the graph. Users accessing one vertex are often interested in adjacent vertices, and by carefully laying out the graph we can make such follow-up queries much faster than an otherwise random request.

We investigate graph layout at two granularities. First, we are interested in *partitioning* the vertices of the graph into a few parts, allowing simpler concurrency and parallelization between multiple workers. Second, we are interested in the relative *placement* of the vertices within a part, ideally with proximate vertices placed near one another to exploit caches at various levels. Partitioning is a very binary separation, two vertices are either in the same part or not, whereas placement is more continuous, in that two vertices can be placed within a spectrum of distances.

3.1 Partitioning and Placement Algorithms

Graph partitioning is a well-studied problem. The objective of a graph partitioning algorithm is to split a graph into smaller sub-graphs such that the number of edges that run between the sub-graphs (*i.e.*, the edge-cut between the partitions) is small. Another important criteria to consider while partitioning a graph is to create balanced partitions, that is, sub-graphs of roughly equal size. The number of vertices alone does not guarantee balanced partitions; for example, even with similar sizes, a partition with more high-degree vertices may become 'hotter' than other partitions. In addition, it is desirable to have a partition algorithm that can run faster, can be parallelized, and can be applied on incremental updates without having to look at the entire graph.

Optimizing all these criteria is a challenging, and unsolved problem. Moreover, different algorithms can have very different behavior on different graphs. Consequently, Grace provides an extensible library of partitioning algorithms, initially stocked with three simple but useful candidates: 1. a hash-based scheme oblivious to the graph structure, 2. a folklore heuristic, based on placing vertices in parts with fewest non-neighbors, and 3. a spectral partitioning algorithm based on the second eigenvector of the normalized Laplacian.

3.1.1 Hash Partitioning

Partitioning a data set based on the hash of the data is well-known. We hash by the vertex ID and distribute the vertices to different partitions. The nice features of hash partitioning are: it is fast; it does not require an entire graph to be loaded into memory and therefore, can be applied on incremental graph updates; it creates well-balanced partitions, and since it does not look at the degree of a vertex, high-degree vertices are mostly uniformly distributed, which reduces the chance of a particular partition becoming overloaded. However, because of its graph-agnostic nature, sub-graphs created using hash partition have high edge-cuts.

3.1.2 Heuristic Partitioning

One folklore heuristic for partitioning repeatedly considers vertices and places them in the part with fewest non-neighbors. All other things being equal, this is very similar to placing them in the part with the most neighbors, but avoids the degenerate case where all vertices join the same part. Instead, there is a tension between choosing an appealing part and balancing the parts.

More formally, for each vertex v with neighbors $N(v)$, the heuristic selects the part P_i minimizing the number of vertices not in $N(v)$: $|P_i \setminus N(v)|$. This number can be easily tracked by maintaining the sizes of each part, and a vector of current assignments of vertices to parts. Evaluating a vertex v requires only looking up the parts of its neighbors, and then subtracting these totals from each part size. The algorithm streams sequentially over the edge file, and only needs to perform random accesses to the vector of current assignments. Despite its simplicity, we find that the heuristic provides significant improvements in our experiments.

3.1.3 Spectral Partitioning

Spectral graph partitioning [29] uses the eigenvectors of the connectivity matrix of the graph to partition nodes.

Despite the intimidating name, it has a fairly simple description: we associate a real value with each vertex, and repeatedly update the value of each vertex to the average of the values of its neighbors. This process provably converges to values reflecting an eigenvector of the connectivity matrix, and intuitively assigns positive values more often to vertices whose neighbors are positive, and less often to those whose neighbors are negative, and vice-versa. Any threshold (we use zero) partitions the graph into two parts, nodes whose values are less than and greater than the threshold, for which we expect fewer edges between the parts than within the parts. One can then repeat the process recursively, until a desired size is reached.

We use hash and heuristic-based partitioning algorithms to create graph parts and spectral partitioning to order the graph vertices in-memory by their associated real values. Even though spectral partitioning can be used for graph partitioning, we did not use it because it is hard to generate balanced partitions and it runs slower than the other two.

On top of vertex placement, we also order the edges within a partition. Edges are first grouped according to their source vertices, following the same vertex ordering, and then within each group (with the same source), edges are ordered by their destination partition. While this placement improves most query performances, we also explore a different edge ordering for iterative computations, which is detailed in the next section.

4 Iterative Execution Platform

On top of the simple querying interface in Grace, we built an execution platform, which can help users program and run highly parallel iterative graph computations, without worrying about complex thread management and system-level optimizations. Similar to Pregel [21], Grace adopts the Bulk Synchronous Parallel (BSP) [30] abstraction and implements a vertex-based data propagation model, which is effective for exploiting parallelism and convenient for programming.

In this programming model, a graph computation consists of a sequence of iterations, separated by global barriers. Conceptually, each iteration has three steps: first, updates from a previous iteration are received; second, user-defined functions are invoked to apply the updates and compute new values for each vertex; and finally, updates are propagated to other vertices for the next iteration. The sequence of iterations are stopped when every vertex – that is, the user-defined function running for every vertex – votes to halt.

Since Grace operates exclusively in a multi-core environment with cache coherent shared memory, propagation of updates simply involves applying the updates directly to other vertices' data. Specific to iterative computations, Grace implements two optimizations: first, updates can be batched per destination partition to minimize data shuttling among cores; second, load from a thread, which handles a 'hot' partition, can be dynamically shed to other threads resulting in balanced computation across cores.

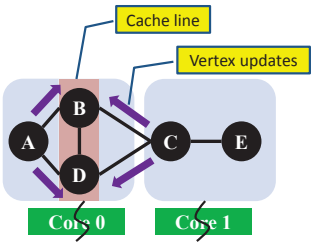


Figure 2: **An Example for Batching Inter-Core Messages.**

4.1 Batching Updates

For a given partitioned graph and a general-purpose iterative computation, Grace has limited opportunities to reduce the updates propagated among partitions. However, by carefully orchestrating *when* updates are propagated, Grace can substantially reduce unnecessary data copies among various cores.

Most modern multi-core architectures present a cache-coherent shared memory. That is, when a core modifies a piece of data, the data is populated on its cache and invalidated on other cores' caches that may have a copy of the data. Since a single cache line can hold multiple data items – which is especially true for graph computations, where a vertex data can be small such as the rank in PageRank or component Id in weakly-connected components – batching updates to all such data items together can minimize the number of cache copies and invalidations. This batching is especially useful in Grace, where vertices can be laid out according to their proximity in the graph.

Figure 2 illustrates a specific example of the above scenario, where batching is beneficial. It presents a graph with two partitions, 0 and 1, each handled by a thread pinned to its core. Consider a case where vertices B and D are placed in contiguous memory addresses and indexed into the same cache line. During an iterative computation, if vertices A and C propagate their updates to B and D, there are at least two possible schedules for applying the updates: in the first schedule, A sends its update to B, then C sends its update to B, which is followed by A's and C's updates to D; in the second schedule, A sends its updates to B and D, which is followed by C's updates to B and D. Assuming that the cache line

with B and D was present in core 0 initially, in the first schedule, B and D are shuttled between partition 0 and 1 for three times, whereas in the second schedule, they are copied only once.

Strictly scheduling threads in a choreographed manner to avoid redundant data copies is difficult and can incur additional overheads. Instead, Grace implements a simple, yet effective memory layout and schedule for propagating updates. Within each partition, all the edges are grouped according to their destination partition, and within each group of destination partition, edges are ordered according to their source vertices' in-memory ordering. This is similar to the edge ordering proposed earlier in Section 3, except that edges are first ordered by destination and then by source. During an iterative computation, whenever updates must be propagated, a thread running in partition i, first selects the edges targeting itself and then selects the next partition in a round-robin fashion (*i.e.*, $(i + 1)\ mod\ N$, where N is the total number of partitions). In our evaluation, we find that batching updates as mentioned above can greatly improve the performance for dense graphs with large average vertex degree.

4.2 Balancing Load

In BSP programming model, iterative computations are separated by global barriers; that is, a partition can execute its next iteration only after all other partitions complete their current iteration. Although such synchronous execution simplifies programming, it introduces a cause for concern. If one partition's execution significantly lags behind others, then the whole computation is delayed proportionally.

In order to maximize the parallelism, it is necessary to balance the load among worker threads such that they are all busy during the entire computation. A standard way of achieving this is by partitioning the data equally among the workers; for example, a graph can be statically partitioned into sub-graphs of equal size. However, there are several drawbacks in static partitioning. First, it is difficult to define 'balance' because it is application-dependent. For example, partitions with same number of vertices may run computations for different time. Balancing the number of edges does not guarantee similar runtime either because not all edges propagate updates during each iteration. Second, enforcing balanced partitioning can degrade partitioning quality with respect to the size of edge-cut and reduce benefits from graph-specific locality.

Instead of statically balancing the sub-graphs, Grace dynamically balances the load by an efficient work sharing mechanism. During an iterative computation, instead of statically assigning all the vertices in a partition to a thread, Grace allows a thread to grab a set of vertices from other partitions, if necessary. Each thread starts processing vertices in its own partition. After all the vertices have been processed in its current partition, a thread grabs a set of vertices from another partition that has not yet been fully processed. For example, if there are two overloaded partitions and three free worker threads, the three threads repeatedly take a portion of vertices from the two partitions until everyone completes.

5 Transactional Graph Updates

The need to support graph updates stems from our initial goal of building a fast graph management system for graphs that may change continuously. Although updates can be applied by simply locking a graph, we chose to implement it with transactions in order to improve the concurrency between graph queries and updates; for example, Grace can run long iterative computations, while concurrently allowing a graph to change.

Grace supports structural changes to a graph. That is, a vertex or an edge can be added or deleted and edge weights can be changed. To update a graph, a thread starts a transaction, issues a set of changes, and finally tries to commit the transaction. The transaction may commit or abort depending on other conflicting changes that may have been applied to the graph. Once a transaction is committed, any new snapshot created from the graph will reflect the changes.

Under the covers, Grace implements transaction using snapshot-isolation [31]. When a transaction is started, Grace creates a consistent, read-only snapshot of the graph and allocates a temporary buffer for storing the updates. Grace ensures that the snapshot is consistent by making its creation atomic with respect to other transactional updates. All reads issued by the transaction are served from its snapshot and changes made by the transaction are logged into the temporary buffer.

During commit, Grace detects conflicts using version numbers. Version numbers can be maintained for each vertex, and although it reduces transaction conflicts, such fine granularity can add a lot of overhead; on the other hand, version can be managed at a large granularity for each graph, which will conflict on every concurrent transaction. Grace finds a middle ground and detects conflicts at the granularity of partitions. When a transaction prepares to commit, it checks if the partitions, which will be changed, were modified since the snapshot was taken. If not, the transaction proceeds to commit. During commit, contents of the temporary buffer are logged into the durable media and then, the changes are applied to the graph. After the transaction commits, the snapshot is deleted.

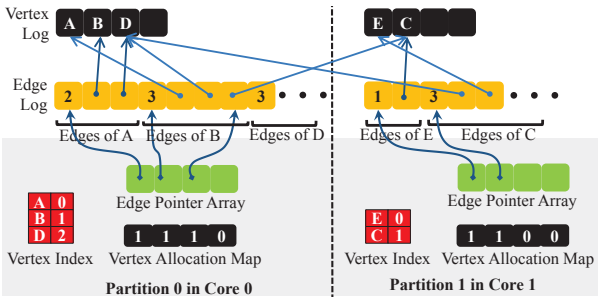


Figure 3: **In-memory Data Structures in Grace.**

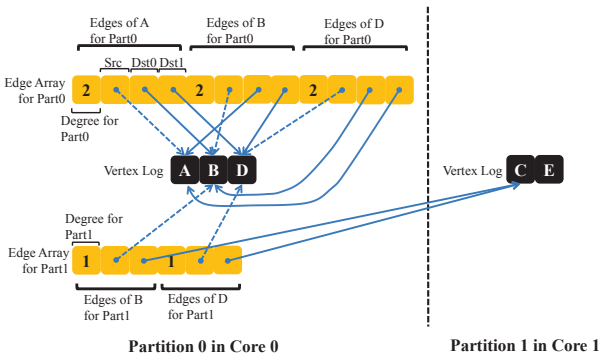


Figure 4: **Data structures of partition-based edge grouping.**

6 Implementation

When implementing Grace, we took careful measures to avoid features and data structures that can affect the performance. Grace is implemented in C++, which gives us the freedom to manipulate the memory layout to our convenience. In this section, we give an overview of the in-memory structures and how they are used to run transactions; finally, we briefly explain the on-disk data.

6.1 In-memory Data Structures

Figure 3 presents an in-memory data organization of the graph shown in Figure 1.

A graph object contains a set of partitions. Within each graph partition, vertices and their edges are stored in separate arrays, which are managed as *in-memory logs* (Vertex Log and Edge Log, in Figure 3). Vertex Log stores per-partition vertex records and the Edge Log stores the edge set – which is the degree and edges of a vertex – of all the vertices in that partition. An edge itself is a composite value consisting of a partition ID and the position of the destination vertex in Vertex Log.

In addition to the Vertex and Edge Logs, Grace creates and manages a few other in-memory structures. An Edge Pointer Array is used to store the position on the Edge Log where a vertex's edges set is stored. For example, in Figure 3, in Partition 0, vertex B's (whose index is 1 in its Vertex Log) edge set is pointed by the contents of Edge Pointer Array at position 1. Grace also maintains an index, mapping a vertex ID to its location on Vertex Log. Finally, Grace uses a Vertex Allocation Map to track whether a position on the Vertex Log contains a valid vertex or not; for example, a vertex can be marked as deleted by clearing the corresponding bit in Vertex Allocation Map.

Grace uses several other data structures, which are not shown in Figure 3 for simplicity. Grace separates any data associated with a vertex (such as the 'rank' in PageRank) and stores them on a separate array, at the same index position as the vertex on the Vertex Log. This separation of data and metadata improves performance considerably, especially for queries that only look at the graph structure. Grace maintains a light-weight bitmap locks for coordinating access to a vertex's data from different partitions.

6.1.1 Implementing Iterative Computations

Iterative computation platform is built on top of the simple APIs exposed by Grace and batching the updates and balancing the load are implemented separately at this layer.

As we detailed earlier, for batching updates, edges are first grouped according to their destination partitions and then within each group, they are ordered by their source vertices. To implement this, each partition uses one edge array for each destination partition as shown in Figure 4. For simplicity, we only show the data structures in partition 0, which has two edge arrays, one for partition 0 and 1. In addition to storing the edges, we also store the number of edges each vertex has for a specific destination partition and a pointer to the vertex's data (dashed arrow). If a vertex does not have any updates to propagate, it can be quickly skipped (based on the number of edges that vertex has on the edge array); however, if a vertex has some updates, its data can be quickly accessed and propagated.

It is important to note that iterative computations are executed on top of graph snapshots and therefore, use separate data structures from the in-memory ones; as a result, iterative computations are unaffected by transactional updates.

6.1.2 Implementing Transactions

Grace uses temporary in-memory buffers for collecting transactional updates and a version vector for a graph, where each element of the vector represents the version of each partition.

Consistent, read-only graph snapshots can be created instantaneously in Grace using copy-on-write tech-

niques. Therefore, at any instant there will be one read-write graph version (on which updates will be applied) and there may be several read-only snapshots. Some of the data structures are shared between all the graph copies, while others are specific to each version. The Vertex and Edge Logs are common across read-write and read-only versions of the graph. Any change made on the graph (such as adding a vertex or modifying an edge set) is appended to the end of the corresponding logs, without affecting other snapshots; that is, no vertex or edge is modified in-place in the log. However, Edge Pointer Array, Vertex Index, and Vertex Allocation Map are unique to different versions of snapshots.

When a new vertex is added, Grace determines the partition where the new vertex should go by rerunning the partition algorithm. Then, the new vertex is appended to the end of the Vertex Log; a copy of the Vertex Index is created, where the new vertex's mapping is included.

If an existing vertex's edge set is modified, the newly modified edge set is appended to the end of the Edge Log; a copy of the Edge Pointer Array is created and its entry corresponding to the modified vertex is changed to point to the new edge set. Finally, if a vertex is deleted, Grace copies the Vertex Allocation Map and clears the corresponding vertex bit in it. In all cases, none of the entries in Vertex Log or Edge Log are updated in place and only the copy-on-write versions of Vertex Index, Edge Pointer Array, and Vertex Allocation Map are modified.

6.2 On-disk Data

Grace's on-disk data is maintained in a fairly straightforward manner. Each partition stores its vertices and edges in respective files. When loading a graph, all the vertices and edges are read parallelly by the partitions, resulting in an overall fast graph loading. In addition to the vertex and edge files, Grace also maintains an on-disk log of committed updates, to recover from crashes.

7 Experiments and Evaluations

We evaluate Grace on two commodity multi-core machines running Windows Server 2008. First machine has 96 GB of memory and four 2.29GHz AMD Opteron 6176 processors, each of which has 12 cores. The other one has 24 GB of memory and two 2.4GHz Intel Xeon E5645 processors, each having 6 cores.

We run representative graph applications such as iterative graph computations, graph traversal algorithms, and online graph queries as benchmarks. For iterative computations, we choose PageRank, Weakly Connected Component (WCC), and Single Source Shortest Path (SSSP). For graph traversal, we select Depth-First Search (DFS) and Breadth-First Search (BFS). Finally, for online queries, we use n-hops queries that retrieve n-hops neighbors' information for a random vertex.

App	Orkut graph			Web graph		
	BDB	Neo4j	Grace	BDB	Neo4j	Grace
PgRk	6,548.7	2,428.0	71.1	6,271.8	31,600.0	280.1
WCC	3,510.9	4,315.0	69.3	8,861.9	48,092.0	671.6
SSSP	700.8	2,000.0	12.4	6,336.7	12,702.2	210.1
BFS	1,227.6	2,732.0	7.6	3,743.2	32,500.0	68.5
DFS	1,329.5	1,882.0	19.1	3,707.1	33,416.5	51.6
q4hop	2,052.7	1,791.2	9.1	4.2	32.0	0.162
q3hop	43.9	52.1	0.636	0.473	15.1	0.02

Table 1: **Comparison With Existing Systems.** This table presents the workload running time (seconds) on BDB, Neo4j, and Grace. PgRk refers to PageRank (5 iterations) and q$n$hop refers to a query for n-hops neighbors. To make query run time measurably long, we randomly pick 8 vertices to query from Orkut graph, and randomly select 128 vertices to query from web graph, which has a smaller average degree.

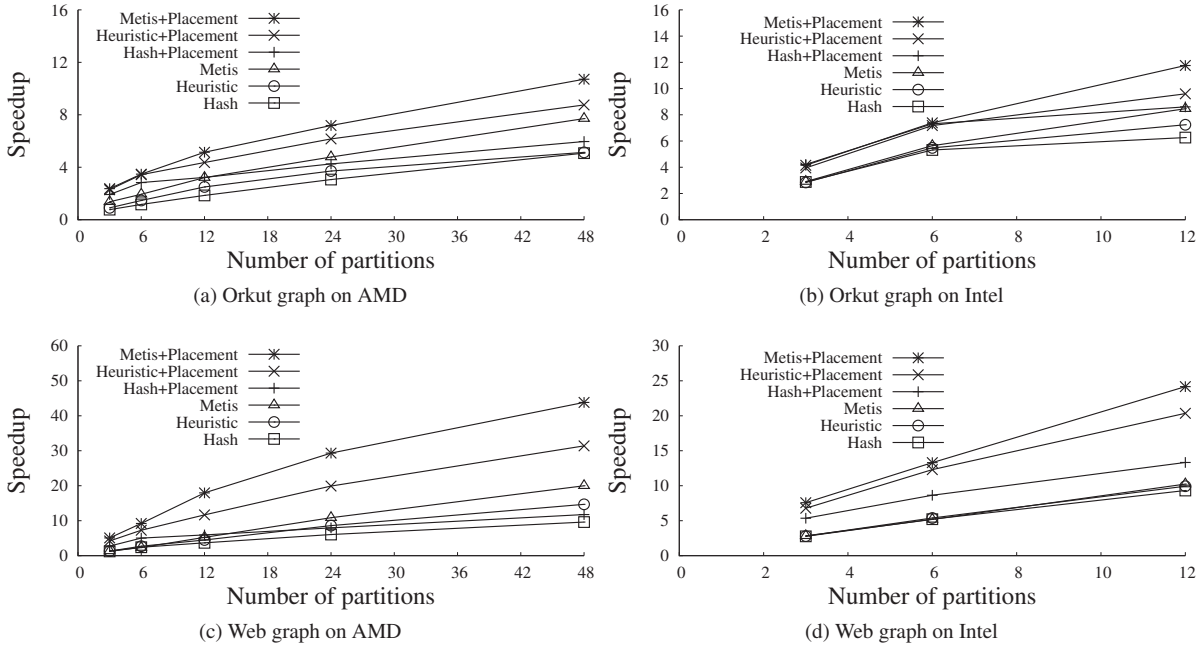
We run our benchmarks on two different graphs: a social network graph and a web graph. For the social network graph, we use a dataset of Orkut which was collected between Oct 3 and Nov 11 2006, and can be publicly accessed [24]. It contains 3,072,441 vertices (users) and 223,534,301 edges. For web graph, we use a dataset from ClueWeb09 [2], which contains the first 10 million English pages crawled during January and February 2009. The corresponding graph consists of 88,519,880 vertices (URLs) and 447,001,255 edges.

In all experiments, applications are run after the whole graphs are loaded into memory and the graphs are kept in memory until the applications finish. Our experiments are designed to answer the following questions. 1. Does Grace outperform existing graph management systems? 2. How effective are the graph-aware and multi-core-specific optimizations? 3. How transactional copy-on-writes impact the runtime? and 4. Does Grace scale to large graphs? We explore these questions in the following sections.

7.1 Comparison with Existing Systems

In order to demonstrate the effectiveness of Grace, we compare it with two existing systems: Berkeley DB (BDB), a widely used open-source key-value store and Neo4j, an open-source transactional graph database implemented in Java.

In BDB, both the key and value are organized as raw bytes. Therefore, to store a graph, we serialize the vertex Id and edges into the key and value payload. For fair comparison, we use the in-memory mode of BDB and set its index as hash table. In Neo4j, we use its vertex and edge creation APIs to import the graphs into its storage. Unlike BDB, Neo4j does not provide an in-memory mode and therefore, we scan the entire graph to preload it into memory before starting applications.



(a) Orkut graph on AMD

(b) Orkut graph on Intel

(c) Web graph on AMD

(d) Web graph on Intel

Figure 5: **Hash vs. Heuristic Partitioning on PageRank**

Since both BDB and Neo4j do not support partitions, we configured Grace to load a graph in to a single partition, where the vertices are rearranged according to their graph proximity, and all three systems use a single thread to run the workloads.

Table 1 compares the execution time of all the applications on BDB, Neo4j, and Grace, running on the AMD machine. The result shows that Grace can perform significantly better than the other two systems – sometimes, up to two orders of magnitude faster – because of its graph-aware optimizations and compact data structures. Other reasons for the poor performance of BDB and Neo4j are: first, search for a vertex following an edge involves an indirect hash-table lookup on vertex Id; second, data structures representing a vertex or an edge are neither compact nor cache-aligned, which leads to larger memory footprint and more cache misses; finally, BDB incurs an additional overhead because it requires extra data copies between its internal key-value structures and the data structures used by applications.

7.2 Optimization Effectiveness

Given the large performance gap between Grace and other systems, we further explore to understand the effectiveness of our optimizations.

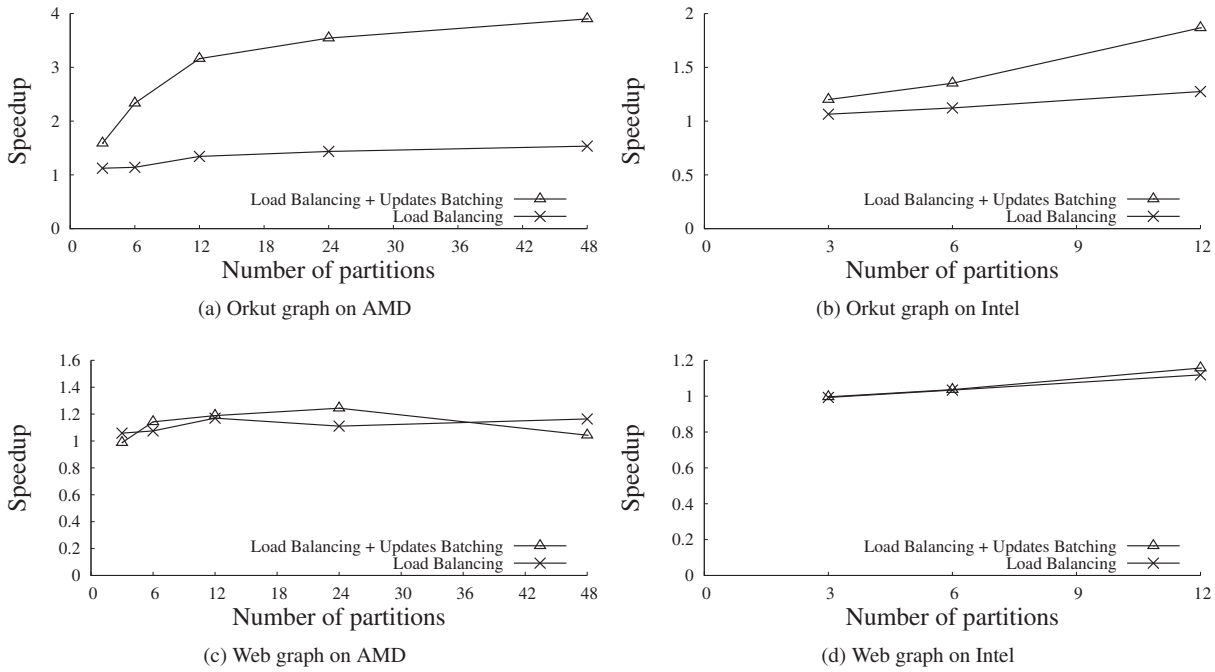
7.2.1 Partitioning and Placement

We run experiments to understand the effectiveness of graph-aware partitioning and vertex placement. For better understanding, we also integrate Metis, a public implementation of multilevel k-way graph partitioning algorithm [5, 19], into Grace's partitioning library for comparison. Figures 5a–5d show the performance of PageRank on Orkut and web graphs on Intel and AMD machines. Each figure plots the speedup from hash, heuristic, and Metis partitioning, with and without vertex reordering, relative to Grace's performance on an unoptimized single partition.

	L1	L2	L3	DTLB
Orkut-Intel	0.46 (6036)	0.50 (5326)	0.44 (1282)	0.31 (999)
Orkut-AMD	0.51 (6825)	0.41 (3815)	0.48 (19469)	0.33 (1118)
Web-Intel	0.23 (7908)	0.27 (12505)	0.20 (2997)	0.24 (6749)
Web-AMD	0.38 (14245)	0.24 (10005)	0.27 (47346)	0.24 (6378)

Table 2: **Relative Reduction in Cache and TLB Miss.** The table shows reduction in cache and TLB miss for PageRank on 12 partitions of Orkut and web graphs when heuristic partitioning and vertex placement optimizations are used. The results are presented relative to the case when only heuristic partitioning is used. Actual number of cache and TLB misses, in millions, is shown in brackets.

First, graph-aware partitioning alone does not improve the performance if the number of graph partitions are fewer. However, the differences between these partitioning techniques grow larger at higher number of partitions. Second, after the vertices and edges are ordered according to their graph-specific locality, both heuristic and Metis partitioning significantly outperform hash partitioning because when a graph is partitioned in a graph-aware manner, the rearrangement algorithm has more opportunities to place a vertex and its neighbors close to



(a) Orkut graph on AMD

(b) Orkut graph on Intel

(c) Web graph on AMD

(d) Web graph on Intel

Figure 6: **Load Balancing and Updates Batching for PageRank**

each other, resulting in better locality.

To better understand the effects of vertex placement, we also measured the cache and TLB miss counts through VTune [7] on Intel machine and CodeAnalyst [3] on AMD machine. Table 2 presents the relative reduction in cache and TLB miss when vertices were rearranged after heuristic partitioning relative to the case when no vertices were rearranged. These results demonstrate that the graph-aware vertex placement can significantly improve the graph data locality.

However, the actual speedup from graph-aware partitioning and placement depends on both the graph and the architecture. Sparse graphs such as the web graph are easier to partition and order, whereas dense graphs such as Orkut are less amenable to vertex layouts. Also, we find that the AMD architecture, with its 48 cores, is more sensitive to cross core communications – perhaps due to the large number of cores – than the 12-core Intel. As a result, we find that the web graph on AMD enjoys a significant speedup from a graph-aware partitioning and placement, whereas Orkut graph on Intel only gains a relatively moderate improvement. Since the Intel and AMD architectures have other differences as well, more investigation is required to understand the reasons behind the different speedups. WCC and SSSP workloads, which are not shown here, also have similar performance curves.

	L1	L2	L3	DTLB	Remote core	Remote chip
Bal	5867	5172	1343	1019	661	1159
Bal+Bat	4057	2603	421	17	204	291

Table 3: **Hardware Event Counts for Load Balancing and Update Batching.** This table shows the hardware event counts for PageRank running on Orkut graph with 12 partitions on Intel machine. It compares load balancing (**Bal**) with both load balancing and updates batching (**Bal+Bat**). **Remote core** counts the number of retired memory load instructions that hit in the L2 cache of a sibling core on the same die. **Remote chip** measures the number of retired memory load instructions that hit the remote processor socket.

7.2.2 Batching Updates and Balancing Load

Next, we focus on understanding the benefits of load balancing and batching updates on iterative computations.

Figures 6a–6d show the additional benefits (or lack thereof) from load balancing and batching, relative to the performance of Grace with heuristic partitioning and placement, for PageRank workload on Orkut and web graphs on both the machines. Each figure shows two curves: the curve 'Load Balancing' represents the benefit just from sharing work among threads and the curve 'Load Balancing + Updates Batching' is the speedup from enabling both the optimizations. As can be noted from the figures, these optimizations perform differently for different graphs and architectures.

First, heuristic partitioning's strength in producing balanced sub-graphs can be observed from the fact that

App	AMD (48 cores)		Intel (12 cores)	
	Orkut	Web	Orkut	Web
PgRk	34.1	36.5	17.9	23.5
WCC	40.4	19.2	14.5	12.5
SSSP	14.8	8.2	6.8	8.2

Table 4: **Maximum Speedup.** This table presents the maximum speedup Grace gets with all optimizations and maximum number of partitions (as supported by the architecture). Speedup is measured relative to a baseline with a single partition and no optimizations.

load balancing strategy did not offer substantial benefits. Second, updates batching is very effective for dense graph such as Orkut, which offers more destination edges for batching and hence shows better performance. Sparse graphs such as web graph do not have sufficient destination edges for updates batching to be effective; however, since the source vertex's data is accessed once for every destination partition (as explained in Section 6.1.1), the extra overhead can sometimes reduce the performance. We noticed similar results for other workloads as well and they are not shown here.

We measure the performance counters for PageRank on Orkut graph on Intel machine to better understand the effects of update batching (we do not measure the numbers for AMD machine since, to the best of our knowledge, it does not provide counters for cross-core access). Table 3 shows that the number of remote core and remote chip accesses significantly decrease when updates are batched. We notice that the cache and TLB miss counts also decrease considerably. We believe that this is mainly caused by the following two reasons: first, since each remote core access implies an L2 cache miss (L2 cache is private for each core) and each remote chip access implies an L2 and L3 cache miss (L3 cache is shared by the cores on the same chip), reduction in remote core or chip accesses will also decrease L2 and L3 cache misses; second, batching could lead to access patterns with better locality because vertices in the same partition are more likely to be in the same cache line than vertices from different partitions.

It is clear that not all optimizations work equally well on all graphs (or architectures); so, it is important to have a system configurable to specific scenarios. Grace provides the flexibility to (manually) choose the right configuration desired by the user. In the future, we can add logic that automatically selects optimizations appropriate for a graph; for example, for sparse graphs, Grace can turn off updates batching.

7.2.3 Overall Speedup

Table 4 presents the maximum speedup Grace achieves for a specific workload-graph-and-architecture combination. The speedup is calculated with respect to a Grace configuration with just one partition and without any other optimization. We can notice speedups up to 40 times (for the case of WCC on Orkut graph on AMD), reducing the runtime of those batch-processed workloads to a few minutes.

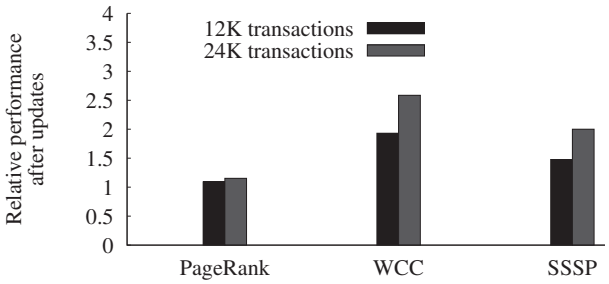


Figure 7: **Impact on Runtime due to Copy-on-Writes.**

7.3 Effect of Transactional Updates on Layout

Copy-on-write is a great mechanism for implementing instantaneous snapshots, consistent updates, and transactions. However, by their very nature, copy-on-writes can disrupt object layouts; specifically, when a data item is changed, it is copied to a different location voiding any careful cache or memory layout optimizations previously made. We evaluate the transactional support in Grace to understand how it affects the in-memory layout and therefore, the performance of workloads.

Figure 7 presents the increase in runtime for all the iterative workloads on Orkut graph, after the graph was updated with transactions; the measurements are presented relative to the runtime on Orkut graph without any changes. Each transaction either randomly deletes a vertex or adds a new vertex with the same set of neighbors as the last deleted one; although, this does not change the conceptual structure of the graph, it does affect Grace's memory layout significantly, where every neighbor and neighbor's edges of a deleted (or added) vertex is copied to the end of the Vertex Log or Edge Log. We vary the number of transactions and estimate the increase in the runtime. We can find that degradation in performance is workload dependent; whereas PageRank suffers only a small increase in runtime, others like WCC and SSSP incur a noticeable drop in performance (up to 2.5 times, for 24K transactions). To overcome such performance drops, Grace can periodically reorder its modified memory layout and regain its original performance.

7.4 Scalability of Grace

Our previous experiments showed that Grace can scale well on large multi-cores and our final experiment measures the scalability of Grace to large graphs. We used a

Graph	PageRank	WCC	SSSP
Hotmail	190.1	109.9	46.5

Table 5: **Scalability to Large Graphs.** This table presents the running time (seconds) of iterative workloads on Hotmail graph.

Hotmail user graph, which is a weighted graph with over 320 million vertices and 575 million edges and ran the iterative workloads on it. Table 5 presents the runtime of iterative computations, which shows that Grace can reduce the runtime of PageRank-like workloads (which we ran for 20 iterations) to close to 3 minutes.

8 Related Work

Systems for Graphs. Early research systems for graphs were built for web graphs, where fast random access is a basic requirement; consequently, systems such as Connectivity Server [9] kept entire graphs in memory and focused on how to compress them efficiently. While early systems ran on a single machine, later systems such as Scalable Hyperlink Store [25] were distributed in design. Unlike these systems, Grace is general-purpose and do not rely on specific graph characteristics. Moreover, none of the early systems focused on multi-core optimizations or transactional updates.

To support efficient graph computations, systems like Pregel [21], Naiad [23], GraphLab [20], and Parallel Boost Graph Library [14, 15] provide a natural graph programming API, which is adopted by Grace. Of all these systems, Grace is more similar to GraphLab as they both operate in memory and run on multi-cores; however, the similarities stop there. Whereas GraphLab is optimized for machine learning algorithms, Grace is general purpose. Grace is unique in how it uses its graph awareness for efficient in-memory graph representations. Moreover, none of these systems, including GraphLab, provide transactional or snapshotting capabilities.

Neo4j [6] is an open-source, single-box graph database. It supports transactions but does not allow graph computations to run concurrently with transactions, unless the computation is wrapped in a big transaction, which makes it likely to be aborted. Moreover, Neo4j's internal in-memory data structures are not optimized for graphs, resulting in poor performance.

Graph-Aware Optimizations. Graph partitioning is a well-studied problem. Its goal is to produce partitions with fewer cross-partition edges. Several offline partitioning algorithms [18, 29] have been proposed, which typically take longer to run. Pujol *et al.* [27] proposed an online partitioning mechanism to obtain good partition quality with a small replication cost. While this works well on distributed systems, – where the cost of accessing a remote partition is far higher than what we observe in multi-cores – we find that a graph-aware partitioning alone does not perform better; however, when accompanied by vertex ordering, graph-aware partitioning works very well on multi-cores.

Imranul *et al.* [17] proposed to leverage the community structure of social graphs to optimize its layout on disk. Similarly, Diwan *et al.* [13] presented a clustering mechanism for tree structure to achieve good locality on disk. Although relevant, these past work optimize for specific graph characteristics (such as social graph or tree), whereas Grace remains general-purpose. Moreover, since Grace keeps an entire graph in memory, it focuses on memory and cache layouts, and does not bother with disk structures.

Multi-Core Optimizations. Since multi-cores are a de facto standard on modern servers, data-parallel computations and key-value stores have been evaluated in them [22, 28]. Recently, many graph algorithms [8, 11, 16] have been specifically designed and implemented for multi-core system.

Jacob *et al.* [26] developed a system runtime out of commodity processors to optimize graph applications. Although they focus on single machine performance, they assume that graph locality is hard to improve and therefore, propose to tolerate memory access latency with massive concurrency. They built a lightweight multithreading library, which accesses memory asynchronously using prefetch instructions. Their technique is orthogonal and can be complementary to Grace.

9 Conclusion

In the last 15 years – perhaps fueled by the stupendous growth of web and, more recently, social networks – systems researchers have been working on special purpose storage and computation platforms for graphs. Grace is another entry in this continuing effort. Grace, with its graph-awareness and optimizations for multi-cores, is unique among them. We plan to extend Grace to run on distributed frameworks, where we hope to apply its current optimizations and discover new ones in those contexts as well.

Acknowledgments

We thank the anonymous reviewers and Pei Cao for their tremendous feedback. We also thank the members of Microsoft Research Silicon Valley for their excellent suggestions and feedback. We are grateful to Frans Kaashoek and Nickolai Zeldovich for their valuable comments on this work, and Sean McDirmid for proof reading.

References

[1] BerkeleyDB. http://www.oracle.com/technetwork/database/berkeleydb/overview/index.html.

[2] Clueweb09. http://lemurproject.org/clueweb09/.

[3] CodeAnalyst. http://developer.amd.com/tools/codeanalyst/Pages/default.aspx.

[4] Hadoop. http://hadoop.apache.org/.

[5] METIS. http://glaros.dtc.umn.edu/gkhome/metis/metis/overview.

[6] Neo4j. http://neo4j.org.

[7] VTune. http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/.

[8] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[9] K. Bharat, A. Broder, M. Henzinger, P. Kumar, and S. Venkatasubramanian. The connectivity server: fast access to linkage information on the Web. In *Proceedings of the 7th international conference on World Wide Web 7*, pages 469–477, 1998.

[10] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[11] G. Cong and K. Makarychev. Optimizing large-scale graph analysis on a multi-threaded, multi-core platform. In *IPDPS*, pages 688–697, 2011.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04*, pages 137–150, 2004.

[13] A. A. Diwan, S. Rane, S. Seshadri, and S. Sudarshan. Clustering techniques for minimizing external path length. In *VLDB'96, September 3-6, Mumbai (Bombay), India*, pages 342–353, 1996.

[14] D. Gregor and A. Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. In *OOPSLA'05*, pages 423–437, New York, NY, USA, 2005. ACM.

[15] D. Gregor and A. Lumsdaine. The parallel bgl: A generic library for distributed graph computations. In *In Parallel Object-Oriented Scientific Computing (POOSC'05)*, 2005.

[16] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, PACT'11*, pages 78–88, 2011.

[17] Imranul Hoque and Indranil Gupta. Social Network-Aware Disk Management. Technical report, University of Illinois at Urbana-Champaign, 2010-12-03.

[18] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20:359–392, December 1998.

[19] G. Karypis, V. Kumar, and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48:96–129, 1998.

[20] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence*, 2010.

[21] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD'10*, pages 135–146, New York, NY, USA, 2010. ACM.

[22] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys'12, Bern, Switzerland*, pages 183–196, 2012.

[23] F. McSherry, R. Isaacs, M. Isard, and D. G. Murray. Naiad: The animating spirit of rivers and streams. Poster Session of Symposium on Operating Systems Principles (SOSP'11), 2011.

[24] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, IMC '07, pages 29–42, New York, NY, USA, 2007. ACM.

[25] M. Najork. The scalable hyperlink store. In *Proceedings of the 20th ACM conference on Hypertext and Hypermedia*, pages 89–98, 2009.

[26] J. Nelson, B. Myers, A. H. Hunter, P. Briggs, L. Ceze, C. Ebeling, D. Grossman, S. Kahan, and M. Oskin. Crunching large graphs with commodity processors. In *Proceedings of the 3rd USENIX conference on Hot topic in parallelism*, HotPar'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[27] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The Little Engine(s) That Could: Scaling Online Social Networks. In *SIGCOMM'10*, New Delhi, India, Aug. 2010.

[28] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA'07*, pages 13–24. IEEE Computer Society, 2007.

[29] D. A. Spielman and S.-H. Teng. Spectral partitioning works: Planar graphs and finite element meshes. In *FOCS*, pages 96–105, 1996.

[30] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.

[31] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2009.

[32] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *OSDI'08, San Diego, California*, pages 1–14, 2008.

MemProf: a Memory Profiler for NUMA Multicore Systems

Renaud Lachaize[†] Baptiste Lepers[†] Vivien Quéma[†]
UJF *CNRS* *GrenobleINP*

Abstract

Modern multicore systems are based on a Non-Uniform Memory Access (NUMA) design. Efficiently exploiting such architectures is notoriously complex for programmers. One of the key concerns is to limit as much as possible the number of remote memory accesses (i.e., main memory accesses performed from a core to a memory bank that is not directly attached to it). However, in many cases, existing profilers do not provide enough information to help programmers achieve this goal.

This paper presents MemProf, a profiler that allows programmers to choose and implement efficient application-level optimizations for NUMA systems. MemProf builds temporal flows of interactions between threads and objects, which help programmers understand why and which memory objects are accessed remotely. We evaluate MemProf on Linux using four applications (FaceRec, Streamcluster, Psearchy, and Apache) on three different machines. In each case, we show how MemProf helps us choose and implement efficient optimizations, unlike existing profilers. These optimizations provide significant performance gains (up to 161%), while requiring very lightweight modifications (10 lines of code or less).

1 Introduction

Multicore platforms are nowadays commonplace and an increasing share of them are based on a Non Uniform Memory Access (NUMA) architecture, i.e., with a set of nodes interacting via interconnect links, each node hosting a memory bank/controller and a group of cores. It is well-known that developing efficient code on such architectures is very difficult since many aspects of hardware resource management (e.g., regarding I/O devices, shared caches and main memory) can strongly impact performance. In this paper, we focus on *remote memory accesses*, i.e., main memory accesses performed on node $N1$ by a thread running on a core from another node $N2$. Remote memory accesses are a major source of inefficiency because they introduce additional latencies in the execution of instructions. These latencies are due to the extra hops required for the communication between a core and a remote memory controller and also possibly to contention on the interconnect links and on the memory controllers [3].

Several techniques have been proposed to reduce the number of remote memory accesses performed by applications running on a NUMA machine, such as memory page duplication [8, 18] or contention-aware scheduling with memory migration [3]. These techniques are "generic" (i.e., application-agnostic): they rely on heuristics and are typically implemented within the OS. However, while useful in some contexts (e.g., co-scheduling), these heuristics (and the online monitoring logic that they require) are not always appropriate since they can possibly hurt the performance of an application, as we show in Section 5. An alternative approach consists in introducing application-level optimizations through lightweight modifications of the application's source code. For example, a programmer can improve the placement of threads and objects by, among other optimizations, modifying the choice of the allocation pool, pinning a thread on a node, or duplicating an object on several memory nodes. Yet, application-level optimization techniques suffer from a significant shortcoming: it is generally difficult for a programmer to determine which technique(s) can be applied to a given application/workload. Indeed, as we show in this paper, diagnosing the issues that call for a specific application-level technique requires a detailed view of the interactions between threads an memory objects, i.e., the ability to determine which threads access which objects at any

[†]Also affiliated with LIG - CNRS UMR 5217.

point in time during the run of an application, and additional information such as the source and target nodes of each memory access. However, existing profilers like OProfile [13], Linux Perf [19], VTune [7] and Memphis [12] do not provide this required information in the general case. Some of them are able to provide this information in the specific case of global static memory objects but these objects often account for a negligible ratio of all remote memory accesses. As an example, for the four applications that we study in this paper, global static memory objects are involved in less than 4% of all remote memory accesses. For the other kinds of objects, the only data provided by the existing profilers are the target memory address and the corresponding program instruction that triggered the access.

In this paper, we present MemProf, the first profiler able to determine the thread and object involved in a given remote memory access performed by an application. MemProf builds temporal flows of the memory accesses that occur during the run of an application. MemProf achieves this result by (i) instrumenting thread and memory management operations with a user library and a kernel module, and (ii) leveraging hardware support from the processors (Instruction-Based Sampling) to monitor the memory accesses. MemProf allows precisely identifying the objects that are involved in remote memory accesses and the corresponding causes (e.g., inefficient object allocation strategies, saturation of a memory node, etc.). Besides, MemProf also provides additional information such as the source code lines corresponding to thread and object creations and destructions. MemProf can thus help a programmer quickly introduce simple and efficient optimizations within a complex and unfamiliar code base. We illustrate the benefits of MemProf on four case studies with real applications (FaceRec [9], Streamcluster [2], Psearchy [4], and Apache [1]). In each case, MemProf allowed us to detect the causes of the remote memory accesses and to introduce simple optimizations (impacting less than 10 lines of code), and thus to achieve a significant performance increase (the gains range from 6.5% to 161%). We also show that these application-specific optimizations can outperform generic heuristics.

The rest of the paper is organized as follows. Section 2 describes a few examples of execution patterns that can benefit from NUMA optimizations and then explains why traditional profilers are not able to pinpoint them. Section 3 presents the main principles of MemProf and how it can be used. Section 4 provides implementation details. Section 5 presents an evaluation of MemProf on four applications. Finally, Section 6 discusses related work and Section 7 concludes the paper.

2 The case for a new profiler

In this section, we present a few examples of optimizations that can be implemented when specific memory access patterns occur in an application. We then explain why existing profilers fail to detect such patterns.

2.1 Application-level NUMA optimizations

We describe a set of three example patterns that negatively impact the performance of applications deployed on NUMA machines. These patterns are inefficient because they increase the number of remote memory accesses performed by an application and their overhead can be significant if they impact objects that are heavily accessed. We review each pattern in turn and explain typical optimizations that can be applied at the application level in order to avoid it.

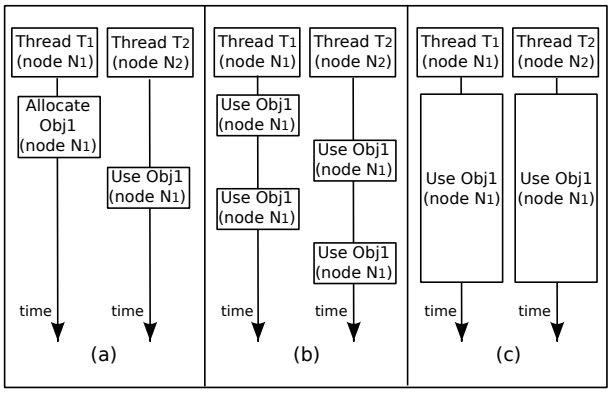


Figure 1: Three memory access patterns that can negatively impact the performance of applications deployed on NUMA machines.

Remote usage after allocation. This pattern (depicted in Figure 1 (a)) occurs when an object is allocated by a thread T_1 on a memory node N_1, and later accessed exclusively (or mostly) by a thread T_2 running on a node N_2. This pattern often occurs in an application with a producer-consumer scheme when the producer and consumer threads are pinned on distinct cores. A simple optimization consists in directly allocating the object on N_2, using NUMA-aware allocation functions. When the application is such that N_2 cannot be determined at the time of the object allocation, another solution consists in migrating the object when T_2 starts to access it, provided that the cost of the data copy can be amortized. Such a migration can be implemented in two ways in the application: (i) using an explicit copy to a buffer allocated on a given node with a NUMA-aware function, or (ii) using a system call to transparently migrate and

remap the set of virtual memory pages that contain the object (e.g., `move_pages()` in Linux).

Alternate remote accesses to an object. In this pattern (depicted in Figure 1 (b)), there are several threads that access a given object over time, but each thread performs its accesses during a disjoint time interval. A first possible optimization consists in adjusting thread placement with respect to the object. Thread placement can be managed in two ways: (i) pinning all the corresponding threads on the node that hosts the object, or (ii) migrating threads over time according to the data they access in a given execution phase. Note that, for some applications, both strategies may create significant load imbalance and thus be inefficient, e.g., when some cores are left idle. A second possible optimization consists in migrating the object close to the thread that currently accesses it (using the migration techniques described in the previous paragraph).

Concurrent remote accesses to an object. In this pattern (depicted in Figure 1 (c)), the object is accessed by several threads during long overlapping time intervals. A first possible optimization to address this pattern consists in pinning all the corresponding threads on the node that hosts the object. Like in the previous case, this optimization may create significant imbalance and thus be inefficient for some applications. A second possible optimization consists in duplicating the object on several memory nodes. This optimization implies to synchronize the multiples object copies (like in a distributed shared memory system) and increases the memory footprint. Consequently, it is mostly applicable to read-only or read-mostly objects whose replication footprint does not exceed the available memory capacity. Finally, if the memory controller of node N_1 is saturated (which can be detected by evaluating the average latencies of memory accesses), two possible optimizations can be considered. The first one consists in balancing the allocation of the different "hottest" objects over multiple memory nodes in order to spread the load. If this optimization cannot be applied, e.g., because the saturation is caused by a few large objects, a second possible optimization consists in interleaving the allocation of each such object over the different memory nodes[1]. Note that, in this case, the optimizations do not necessarily decrease the number of remote memory accesses but allow keeping the corresponding latencies as low as possible by avoiding link or controller saturation.

[1] An application programmer can typically introduce the latter optimization at the page granularity using a kernel API (such as `mbind()` with the `MPOL_INTERLEAVE` flag in Linux).

2.2 Limitations of existing profilers

This section studies whether existing profilers can help detecting inefficient patterns such as the ones described in Section 2.1. We show below that these profilers are actually not able to do so in the general case, because they cannot precisely determine whether two threads access the same object (in main memory) or not[2].

Using existing profilers, a developer can determine, for a given memory access performed by a thread, the involved virtual and physical addresses, as well as the corresponding source code line (e.g., a C/C++ statement) and assembly-level instruction, and the function call chain. In order to extract inefficient thread/memory interaction patterns from such a raw trace, a programmer has to determine if two given individual memory accesses actually target the same object instance or not.

In the case of global statically allocated objects, the answer can be found by analyzing the information embedded in the program binary and the system libraries, from which the size and precise virtual address range of each object can be obtained. This feature is actually implemented by tools like VTune and Memphis. Unfortunately, according to our experience with a number of applications, this kind of objects only account for a very low fraction of the remote memory accesses (e.g., less than 4% in all applications studied in Section 5). In the more general case, i.e., with arbitrary kinds of dynamically-allocated objects, the output of existing profilers (addresses and code paths) is not sufficient to reach a conclusion, as explained below.

First, existing profilers do not track and maintain enough information to determine the enclosing object instance corresponding to a given (virtual or physical) memory address. Indeed, as the lifecycle of dynamic objects is not captured (e.g., dynamic creations/destructions of memory mappings or application-level objects), the address ranges of objects are not known. Moreover, a given (virtual or physical) address can be reused for different objects over time. In addition, virtual-to-physical mappings can also evolve over time (due to swap activity or to page migrations) and their lifecycle is not tracked either.

Second, the additional knowledge of the code path (function call chain and precise instruction) that causes a remote memory access is also insufficient to determine if several threads access the same memory object. Some applications are sufficiently simple to determine the accessed object using only the code path provided by existing profilers. However, in practice, we found that this was not the case on any of the applications that we studied. In general, the code path is often helpful to deter-

[2] From the application-level perspective, these *accesses* correspond to object allocation, destruction, as well as read or write operations.

mine the object *type* related to a given remote memory access, but does not allow pinpointing the precise object *instance* being accessed. Indeed, the internal structure and workloads of many applications are such that the same function is successively called with distinct arguments (i.e., pointers to distinct instances of the same object type), and only a subset of these invocations causes remote memory accesses. For instance, in Section 5.2, we show the example of an application (FaceRec) that processes nearly 200 matrices, and where only one of them is involved in a large number of remote memory accesses.

3 MemProf

In this section, we present MemProf, the first NUMA profiler allowing the capture of interactions between threads and objects. More precisely, MemProf is able to associate remote memory accesses with memory-mapped files, binary sections thread stacks, and with arbitrary objects that are statically or dynamically allocated by applications. This section is organized as follows: we first give an overview of MemProf. Second, we describe the output provided by MemProf. Finally, we describe how MemProf can be used to detect patterns such as the ones presented in Section 2.1.

3.1 Overview

MemProf aims at providing sufficient information to find and implement appropriate solutions to reduce the number of remote memory accesses. The key idea behind MemProf is to build temporal flows of interactions between threads and in-memory objects. Intuitively, these flows are used to "go back in an object's history" to find out which and when threads accessed the object remotely. Processing these flows allows understanding the causes of remote memory accesses and thus designing appropriate optimizations.

MemProf distinguishes five types of objects that we have found important for NUMA performance troubleshooting: global statically allocated objects, dynamically-allocated objects, memory-mapped files, sections of a binary mapped by the operating system (i.e., the main binary of an application or dynamic libraries) and thread stacks. MemProf associates each memory access with an object instance of one of these types.

MemProf records two types of flows. The first type of flow represents, for each profiled thread, the timeline of memory accesses performed by this thread. We call these flows Thread Event Flows (TEFs). The second type of flow represents, for each object accessed in memory, the timeline of accesses performed on the object. We call these flows Object Event Flows (OEFs).

These two types of flows give access to a large number of indicators that are useful to detect patterns such as the ones presented in Section 2.1: the objects (types and instances) that are accessed remotely, the thread that allocates a given object and the threads that access this object, the node(s) where an object is allocated, accessed from and migrated to, the objects that are accessed by multiple threads, the objects that are accessed in a read-only or in a read-mostly fashion, etc. Note that different views can be extracted from MemProf's output, i.e., either focused on a single item (thread/object) or aggregated over multiple items. In addition, the temporal information can be exploited to detect some specific phases in an application run, e.g., read/write phases or time intervals during which a memory object is accessed with a very high latency (e.g., due to the intermittent saturation of a memory controller).

In the remainder of this section, we first describe the output of MemProf. We then provide details on the way to use MemProf.

3.2 Output

MemProf builds one Thread Event Flow (TEF) per profiled thread T. An example of TEF is given in Figure 2 (a). The TEF of a given thread T contains a list of "object accesses"; each "object access" corresponds to a main memory access performed by T. The "object accesses" are organized in chronological order inside the TEF. They contain: (i) the node from which the access is performed, (ii) the memory node that is accessed, (iii) a reference to the Object Event Flow of the accessed object, (iv) the latency of the memory access, (v) a boolean indicating whether the access is a read or a write operation, and (vi) a function call chain. The TEF of a given thread T also contains some additional metadata: the PID of T and the process binary filename. These metadata allow computing statistics about threads of the same process and the threads of a common application.

Object Event Flows (OEFs) provide a dual view of the information contained in the TEFs. MemProf builds one OEF per object O accessed in memory. An example of OEF is given in Figure 2 (b). The OEF of an object O is composed of "thread accesses", ordered chronologically. Each "thread access" corresponds to a memory access to O. The "thread accesses" store similar information as the one found in "object accesses". The only difference is that instead of containing a reference to an OEF, an "object access" contains a reference to the TEF of the thread accessing O. The OEF of a memory object O also contains metadata about O: the type of the object (among the 5 types described in Section 3.1), the size of the object, the line of code where the object was allocated or declared. Besides, there are some additional metadata for a

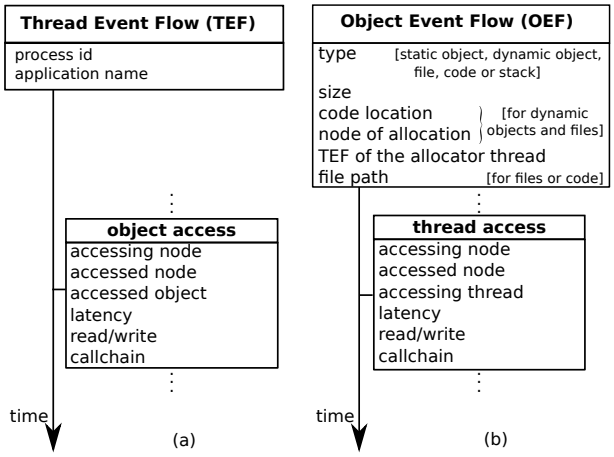


Figure 2: The two types of flows built by MemProf. There is (a) a Thread Event Flow per profiled thread, and (b) an Object Event Flow per object accessed during the profiling session.

dynamically-allocated object, as depicted in Figure 2.

3.3 Using MemProf

MemProf provides a C API to process the TEFs and OEFs. This API provides functions to process events and metadata contained in the TEFs and OEFs. Using this API, it is possible to compute statistics about a single thread or object, or about a group of threads or objects. Due to lack of space, we do not provide a detailed description of the API. Rather, we provide a simple example of script that can be written using this API in Figure 3. This script computes the average time between two memory accesses performed by distinct threads on an object. Such a script can be used, for instance, to estimate the relevance of implementing a memory migration optimization.

MemProf provides a set of generic scripts whose output is often sufficient for understanding the symptoms of an application, e.g., ratio and number of remote memory accesses, list of the most accessed object types, access patterns corresponding to an object type or to a specific object instance.

4 Implementation

In this section, we present the implementation of MemProf for Linux. MemProf performs two main tasks, illustrated in Figure 4. The first task (online) consists in collecting events (thread creation, object allocation, memory accesses, etc.) that are then processed by the second task (in an offline phase), which is in charge of constructing the flows (TEFs and OEFs). We review each task in turn.

```
oef o = ...;
thread_acces a;
u64 last_tid = 0, last_rdt = 0;
u64 nb_tid_switch = 0;
u64 time_per_tid = 0;
foreach_taccess(o, a) {
  if(a.tid == last_tid)
    continue;
  nb_tid_switch++;
  time_per_tid += a.rdt - last_rdt;
  last_tid = a.tid;
  last_rdt = a.rdt;
}
if (nb_tid_switches)
  printf("Avg time: %lu cycles (%lu switches)\n",
    time_per_tid/nb_tid_switch, nb_tid_switch);
```

Figure 3: A script computing the average time between two memory accesses by distinct threads to an object.

4.1 Event collection

The event collection task consists in tracking the life cycle of objects and threads, as well as the memory accesses.

Object lifecycle tracking. MemProf is able to track the allocation and destruction of different types of memory objects, as decribed below.

MemProf tracks the lifecycle of dynamically allocated memory objects and memory-mapped files by overloading the memory allocation functions (`malloc`, `calloc`, `realloc`, `free`, `mmap` and `munmap`) called by the threads that it profiles. MemProf can also be adapted to overload more specialized functions when an application does not use the standard allocation interfaces. Function overloading is performed by linking the profiled applications with a shared library provided by MemProf, through the dynamic linker's `LD_PRELOAD` and `dlsym` facilities[3].

MemProf tracks the lifecycle of code sections and global static variables with a kernel module that overloads the `perf_event_mmap` function. This function is called on every process creation, when the binary and libraries of the process are mapped in memory. It provides the size, virtual address and "content" (e.g., file name) of newly mapped memory zones.

For each kind of object, MemProf stores the virtual address of the created or destroyed object. It also stores a timestamp, the tid of the calling thread, the CPUID on which the function is called and a callchain. The timestamp is required to determine the lifecycle of memory objects. The tid is necessary to know in which virtual address space the object is allocated. The CPUID

[3] MemProf can thus work on applications whose source code is not available. However, its insight may be somewhat restricted in such a case (e.g., given the lack of visibility over the accessed object types).

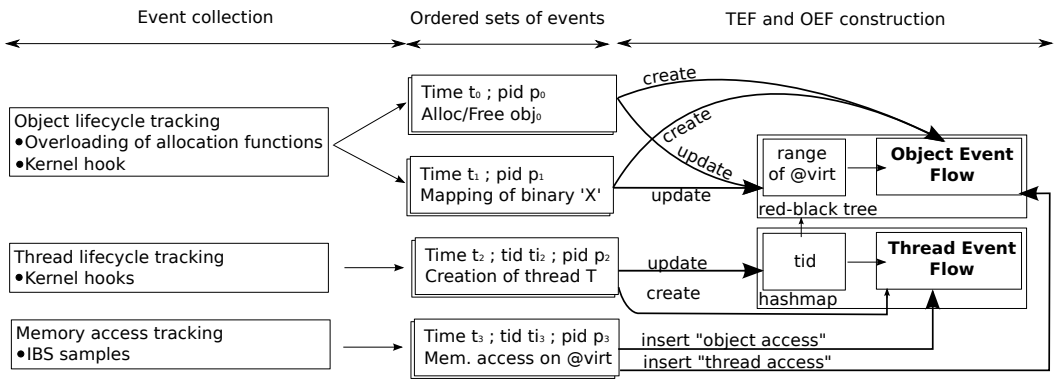


Figure 4: Implementation of MemProf. MemProf performs two tasks: event collection (online) and flow construction (offline). Due to space restrictions, we only present the most important fields of the collected events.

is needed for the corresponding OEF metadata. The callchain is required to find the line of code where the object was allocated.

Thread lifecycle tracking. In order to detect thread (and also stack) creations and destructions, MemProf overloads two kernel functions: `perf_event_task` and `perf_event_comm`, which are called upon such events. The `perf_event_task` function provides the thread id of newly created/destroyed threads, and the `perf_event_comm` function provides the name of newly created threads. MemProf records these metadata and associates them with a timestamp and a pid.

Memory access tracking. Memory accesses are tracked using Instruction Based Sampling (IBS) [5], a profiling mechanism introduced by the "AMD Family 10h" line of processors[4]. It works at the microarchitecture level by periodically selecting a random instruction and recording performance information about its execution. An interrupt is generated when the selected instruction has been fully executed. This interrupt is caught by the kernel module and the IBS sample is saved in a per-CPU buffer. For instructions that reference a memory location, the processor collects the virtual and physical address of the location, the number of clock cycles required to fetch the data, the level where the data was found in the memory hierarchy (in one of the caches, in the local DRAM bank or in a remote DRAM bank), and the access type (read or write). In addition to the information provided by the IBS registers, the kernel module also saves a timestamp, the CPUID of the core, as well as the thread id and stack boundaries of the executing thread.

[4]IBS is only available for AMD processors but a similar hardware-level profiling mechanism, called PEBS [16], exists for Intel processors. MemProf's kernel module could easily be extended to leverage PEBS.

4.2 TEF and OEF construction

Once the events have been collected, MemProf builds (offline) the OEF and TEF of the profiled application(s). As illustrated in Figure 4, the events are first ordered by timestamp. MemProf then iterates over the ordered list of events and builds the flows as follows. MemProf creates a new TEF each time a new thread is created and an OEF each time a new object is allocated. MemProf stores (i) the TEFs in a hashmap indexed by the tid of their thread and (ii) the OEFs in a per-process red-black tree structure. This red-black tree allows finding if a virtual address corresponds to a previously allocated object (e.g., if an object O is allocated on the virtual address range $[0x5 - 0x10]$, then a request for the address $0x7$ will return the OEF of O). When an object is destroyed, its OEF is removed from the red-black tree.

For each memory access to a given virtual address, MemProf (i) creates a "thread access" and inserts it in the TEF of the thread that performed the memory access, and (ii) searches in the red-black tree for the OEF that corresponds to the accessed virtual address; if it exists, MemProf creates an "object access" and inserts it in the OEF. Note that in most cases, the OEF exists in the red-black tree; the only exceptions are (i) when a memory access targets the stack of a thread, and (ii) when the physical address is outside the valid range of physical addresses[5]. MemProf assigns all memory accesses performed on stacks to a single (per-process) OEF representing all stacks. Indeed, we observed in practice that stacks represent only a small percentage of remote memory accesses and it is thus sufficient to regroup all memory accesses performed on stacks in a single OEF. Moreover, MemProf ignores physical addresses that are outside the valid range of physical addresses.

[5]The hardware creates IBS samples with such addresses when it monitors special assembly instructions like `rdtscll`, which is used to read the timestamp counter.

5 Evaluation

In this Section, we evaluate MemProf using four applications: FaceRec, Streamcluster, Apache and Psearchy. The first three applications perform a significant number of remote memory accesses. The last one performs fewer remote memory accesses but is memory-intensive. For each application, we first try to optimize the application using the output of existing profilers, i.e., instruction-oriented profilers like Perf and data-oriented profilers like Memphis. We show that the latter do not give precise insights on what and how to optimize the application. We then profile the application with MemProf and show that it allows precisely pinpointing the causes of remote memory accesses and designing appropriate optimizations to mitigate them. These optimizations are very simple (less than 10 lines of code) and efficient (the gains range from 6.5% to 161%). Finally, we conclude this section by a study of the overhead induced by MemProf.

5.1 Testbed

We perform the evaluation using three NUMA-machines (with 16, 24 and 48 cores respectively) presented below. They all run the Linux kernel v2.6.35 and eGlibc v2.11.
Machine A has 4 AMD Opteron 8380 processors clocked at 2.5GHz with 4 cores in each (16 cores in total), 32GB of RAM and 20 1Gb/s Ethernet ports. It features 4 memory nodes (i.e., 4 cores and 8GB of RAM per node) interconnected with HyperTransport 1.0 links.
Machine B has 2 AMD Opteron 6164 HE processors clocked at 1.7 GHz with 12 cores in each (24 cores in total) and 48 GB of RAM. It features 4 memory nodes (i.e., 6 cores and 12 GB of RAM per node) interconnected with HyperTransport 3.0 links.
Machine C has 4 AMD Opteron 6174 processors clocked at 2.2 GHz with 12 cores in each (48 cores in total) and 256 GB of RAM. It features 8 memory nodes (i.e., 6 cores and 32 GB of RAM per node) interconnected with HyperTransport 3.0 links.

5.2 FaceRec

FaceRec is a facial recognition engine of the ALPBench benchmark suite [9]. We use the default workload included in the suite. FaceRec performs 63% of its memory accesses on remote memory. We first try to optimize FaceRec using existing profilers. We obtain a performance improvement ranging from 9% to 15%. We then try to optimize FaceRec using MemProf. We obtain a performance improvement ranging from 16% to 41%.

Optimization using existing profilers. Instruction-oriented profilers allow understanding that most remote memory accesses are performed in one function (see the output of the Perf profiler, presented in Table 1). This function takes two matrices as arguments and performs a matrix multiplication. It is called on all matrices manipulated by the application (using MemProf, we learn that 193 matrices are allocated during a run of the default ALPBench workload). Instruction-oriented profilers do not allow determining which matrices induce large numbers of remote memory accesses.

% of total remote accesses	Function
98	transposeMultiplyMatrixL
0.08	malloc

Table 1: Functions performing most remote memory accesses in FaceRec.

Data-oriented profilers show that no remote memory access is performed on global statically allocated objects. Moreover, they allow gathering the list of virtual addresses that are accessed remotely, together with the ratio of remote memory accesses that each virtual address accounts for. Nevertheless, it is not possible to determine if a range of accessed addresses represents one or more matrices. Indeed, FaceRec allocates matrices of different sizes, and each range of virtual addresses can be used to store different matrices during different time intervals (MemProf actually shows that FaceRec allocates several matrices on the same virtual address ranges). Consequently, existing profilers do not allow understanding which matrices induce many remote memory accesses.

The optimizations that can be envisioned using the output of existing profilers are: (i) duplicating all matrices on all nodes, or (ii) interleaving the memory allocated for all matrices on all nodes. Both optimizations require the developer to retrieve all places in the code where matrices are allocated. We did not implement the first optimization because it requires writing complex code to synchronize the state of matrices whenever they are updated. Moreover, we know (using MemProf) that this optimization will not induce good performance. Indeed, MemProf shows that some matrices are often updated, and thus that the synchronization code would be frequently triggered. We tried the second optimization, which is simple to implement: it consists in replacing the calls to `malloc` with calls to `numa_alloc_interleaved`. This optimization induces performance improvements of 15%, 9% and 13% on respectively Machines A, B and C. Note that this optimization increases the number of remote memory accesses, but decreases the contention on one of the memory nodes, hence the performance improvement.

Optimization using MemProf. MemProf points out that most remote memory accesses are performed on a single matrix (see Table 2). This explains why the optimiza-

tion presented in the previous paragraph induced a performance improvement: it decreases the contention on the memory node hosting this matrix. Using the OEF of this matrix, we observe that, contrarily to some other matrices, this matrix is written only once and then accessed in read-only mode by a set of threads[6]. We leverage this observation as follows: we optimize FaceRec by duplicating this matrix on all nodes after its initialization. As the matrix is never updated, we did not have to write any synchronization code. The matrix occupies 15MB of memory. The footprint overhead of this optimization is thus 45MB on machines A and B (4 nodes) and 105MB on machine C (8 nodes). The implementation of the matrix duplication only required 10 lines of code. We simply had to modify the `readAndProjectImages` function that initializes this matrix so that it allocates one matrix per node. For simplicity, we stored the various pointers to the matrices in a global array. Threads then choose the appropriate matrix depending on the node they are currently running on. With this optimization, FaceRec only performs 2.2% of its memory accesses on remote memory (63% before the optimization). This results in performance improvements of respectively 41%, 26% and 37% on Machines A, B and C.

% of total remote accesses	Object
98.8	s->basis(csuCommonSubspace.c:455)
0.2	[static objects of libc-2.11.2.so]

Table 2: Objects remotely accessed in FaceRec.

5.3 Streamcluster

Streamcluster is a parallel data-mining application included in the popular PARSEC 2.0 benchmark suite [2]. Streamcluster performs 75% of its memory accesses on remote memory. We first try to optimize Streamcluster using existing profilers and obtain a performance improvement ranging from 33% to 136%. We then try to optimize Streamcluster using MemProf and obtain an improvement ranging from 37% to 161%. This means that Streamcluster is an application for which existing profilers provide enough information to successfully optimize the application, but that MemProf is able to provide details that can be exploited to implement the optimization slightly more efficiently.

Optimization using existing profilers. Instruction-oriented profilers allow understanding that most remote memory accesses are performed in one function (see the output of the Perf profiler, presented in Table 3). This function takes two points as parameters (`p1` and `p2`) and computes the distance between them. The remote accesses are done by the following line of code:

```
result += (p1.coord[i]-p2.coord[i])*(p1.coord[i]-p2.coord[i]).
```

An analysis of the assembly code shows that remote memory accesses are performed on the `coord` field of the points. It is nevertheless not possible to know if all points or only part of them induce remote memory accesses. Instruction-oriented profilers also allow understanding that one of the memory nodes is more loaded than others (i.e., memory accesses targeting this node have a higher latency).

% of total remote accesses	Function
80	dist
18	pspeedy
1	parsec_barrier_wait

Table 3: Functions performing most remote memory accesses in Streamcluster.

Data-oriented profilers show that less than 1% of the remote memory accesses are performed on global statically allocated data. Moreover, they show that threads remotely access the same memory pages. This information is not sufficient to understand if some "points" are more frequently remotely accessed than others (as was the case with matrices in the FaceRec application), nor to understand if threads share data, or if they access different objects placed on the same page.

Several optimizations can be proposed: (i) memory duplication, (ii) memory migration, or (iii) memory interleaving. The first one performs poorly if objects are frequently updated. The second one performs poorly if objects are simultaneously accessed by various threads. As we have no information on these two points, and due to the fact that these two optimizations are complex to implement, we did not try them. The third possible optimization makes sense because one node of the system is saturated. Interleaving the memory allows spreading the load of memory accesses on all memory nodes, which avoids saturating nodes. The optimization works as follows: we interleave all the dynamically allocated memory pages thanks to the `numactl` utility available in recent Linux distributions. With this optimization, Streamcluster performs 80% of its memory accesses on remote memory (75% before optimizing), but memory accesses are evenly distributed on all nodes. This optimization improves performance by respectively 33%, 136% and 71% on Machines A, B and C.

Optimization using MemProf. MemProf shows that most remote memory accesses are performed on one array (see Table 4). Note that a set of objets accounts for 14% of remote memory accesses, but they are not represented in Table 4, as each object individually accounts for less than 0.5% of all remote memory accesses. The

[6] As MemProf only samples a subset of the memory accesses, we checked this fact via source code inspection.

analysis of the OEF of the array shows that it is simultaneously read and written by many threads. The reason why MemProf allows pinpointing an array, whereas existing profilers point out the `coord` fields is simple: the `coord` fields of the different points are pointers to offsets within a single array (named "`block`" in Table 4). MemProf also outputs that the array is allocated on a single node, and that the latency of memory accesses to this node is very high (for instance, approximately 700 cycles on Machine B). Consequently, to optimize Streamcluster, we chose to interleave the memory allocated for this array on multiple nodes. This improves performance by respectively 37%, 161% and 82% for Machines A, B and C. As expected, this optimization does not decrease the ratio of remote memory accesses (75%), but it drastically reduces the average memory latency to the saturated node (430 cycles on Machine B). Note that, using MemProf, the optimization is more efficient: this comes from the fact that only a subset of the memory is interleaved: the memory that is effectively the target of remote memory accesses.

It is important to note that MemProf also gives information about the potential benefits of the memory duplication and memory migration techniques discussed above. Indeed, MemProf shows that the array is frequently updated (i.e., there is a high ratio of write accesses), which indicates that memory duplication would probably perform poorly. Moreover, the analysis of the OEF of the array shows that it is simultaneously accessed by several threads, which indicates that memory migration would also probably perform poorly.

% of total remote accesses	Object
80.9	float *block(streamcluster.cpp:1852)
4.1	points.p(streamcluster.cpp:1865)
1	[stack]

Table 4: Objects remotely accessed in Streamcluster.

5.4 Psearchy

Psearchy is a parallel file indexer from the Mosbench benchmark suite [4]. Psearchy only performs 17% of its memory accesses on remote memory but it is memory intensive: it exhibits a high ratio of memory accesses per instruction. We first try to optimize Psearchy using existing profilers and do not obtain any performance gain (the only straightforward optimization to implement yields a 14-29% performance decrease). We then try to optimize Psearchy using MemProf and obtain a performance improvement ranging from 6.5% to 8.2%.

Optimization using existing profilers. Instruction-oriented profilers allow finding the functions that perform most remote memory accesses (see the output of the Perf profiler, presented in Table 5). The first function, named `pass0`, aims at parsing the content of the files to be indexed. The second function is a sort function provided by the libc library. In both cases, we are not able to determine which objects are targeted by remote memory accesses. For instance, when looking at the assembly instructions that induce remote memory accesses in the `pass0` function, we are not able to determine if the memory accesses are performed on the file buffers, or on the structures that hold the indexed words. Indeed, the assembly instructions causing remote memory accesses are indirectly addressing registers and we are not able to infer the content of the registers by looking at the code.

% of total remote accesses	Function
45	pass0
40	msort_with_tmp
7.7	strcmp
3	lookup

Table 5: Functions performing remote memory accesses in Psearchy.

Data-oriented profilers show that less than 1% of the remote memory accesses are performed on global statically allocated data. As in the case of FaceRec and Streamcluster, Data-oriented profilers show that threads access the same memory pages. We do thus consider the same set of optimizations: (i) memory duplication, (ii) memory migration, or (iii) memory interleaving. For the same reason as in Streamcluster, we do not try to apply the first two optimizations (the analysis performed with MemProf in the next paragraph validates this choice). The third possible optimization does not make sense either. Indeed, contrarily to what we observed for Streamcluster, no memory node is saturated. As this optimization is very simple to implement, we nonetheless implemented it to make sure that it did not apply. It decreases the performance by respectively 22%, 14% and 29% for Machines A, B and C.

Optimization using MemProf. MemProf points out that most remote memory accesses are performed on many different objects. We also observe that these objects are of three different types. We give in Table 6 the percentage that each type of object accounts for with respect to the total number of remote memory accesses. More interestingly, we observe, using the OEFs of all objects, that each object is accessed by a single thread: the thread that allocated it. This means that threads in Psearchy do not share objects, contrarily to what we observed in FaceRec and Streamcluster[7]. This observation is important for several reasons. First, it implies that memory duplication and memory migration

[7] As MemProf only samples a subset of the memory accesses, we checked this fact via source code inspection.

are not suited to Psearchy. Second, it allows understanding why threads — although not sharing objects — all access the same memory pages: the reason is that all objects are allocated on the same set of memory pages. Using this information, it is trivial to propose a very simple optimization: forcing threads to allocate memory on the node where they run. As threads are not sharing objects, this should avoid most remote memory accesses. We implemented this optimization using the `numa_alloc_local` function. With this optimization, less than 2% of memory accesses are performed on remote objects and the performance increases by respectively 8.2%, 7.2% and 6.5% on Machines A, B and C.

% of total remote accesses	Type of object
42	ps.table(pedsort.C:614)
38	tmp(qsort_r+0x86)
10	ps.blocks(pedsort.C:620)

Table 6: Types of the main objects that are remotely accessed in Psearchy.

5.5 Apache/PHP

Apache/PHP [1] is a widely used Web server stack. We benchmark it using the Specweb2005 [15] workload. Because machines B and C have a limited number of network interfaces, we could not use them to benchmark the Apache/PHP stack under high load. On machine A, we observe that the Apache/PHP stack performs 75% of its memory accesses on remote memory. We first try to optimize the Apache/PHP stack using the output of existing profilers. We show that we are not able to precisely detect the cause of remote memory accesses, and that, consequently, all our optimizations fail. We then try to optimize the Apache/PHP stack using MemProf. We show that MemProf allows precisely pinpointing the two types of objects responsible for most remote memory accesses. Using a simple optimization (less than 10 lines of code), we improve the performance by up to 20%.

Optimization using existing profilers. Instruction-oriented profilers allow finding the functions that perform most remote memory accesses (see the output of the Perf profiler, presented in Table 7). No function stands out. The top functions are related to memory operations (e.g., `memcpy` copies memory zones) and they are called from many different places on many different variables. It is impossible to know what they access in memory.

Data-oriented profilers show that less than 4% of the remote memory accesses are performed on global statically allocated data. They also show that many threads remotely access the same set of memory pages. Finally, they show that some pages are accessed at different time intervals by different threads, whereas other pages are simultaneously accessed by multiple threads.

% of total remote accesses	Function
5.18	memcpy
2.80	_zend_mm_alloc_int
1.72	zend_hash_find

Table 7: Top functions performing remote memory accesses in the Apache/PHP stack.

Several optimizations can be tried on Apache/PHP, based on the previously described observations. The first observation we tried to leverage is the fact that some pages are accessed at different time intervals by different threads. Because we did not know which objects are accessed, we designed an application-agnostic heuristic in charge of migrating pages. We used the same heuristic as the one recently described by Blagodurov et al. [3]: every 10ms a daemon wakes up and migrates pages. More precisely, the daemon sets up IBS sampling, *"reads the next sample and migrates the page containing the memory address in the sample along with K pages in the application address space that sequentially precede and follow the accessed page"*. In [3], the value of K is 4096. Unfortunately, using this heuristic, we observed a slight decrease in performance (around 5%). We tried different values for K, but without success. This is probably due to the fact that we use a different software configuration, with many more threads spawned by Apache.

The second observation we tried to leverage is the fact that some pages are simultaneously accessed by multiple threads. As we did not expect Apache threads to share memory pages (each thread is in charge of one TCP connection and handles a dedicated HTTP request stream), we thought there could be a memory allocation problem similar to the one encountered in Psearchy, where threads allocate data on a remote node. We did thus try to use a NUMA-aware memory allocator, i.e., we replaced all calls to `malloc` with calls to `numa_alloc_local`. This did not impact the performance. Although no memory node was overloaded, we also tried to interleave the memory using the `numactl` utility. This optimization did not impact the performance either. Finally, we thought that the problem could be due to threads migrating from one memory node to another one, thus causing remote memory accesses. Consequently, we decided to pin all Apache threads and all PHP processes at creation time on the different available cores, using a round-robin strategy. To pin threads and processes we used the `pthread_set_affinity` function. This optimization had a negligible impact on performance (2% improvement).

Optimization using MemProf. MemProf points out that

Apache performs most of its remote memory accesses on two types of objects: variables named `apr_pools`, that are all allocated in a single function, and the pointer relocation table (PLT). The PLT is a special section of binaries mapped in memory. It is used at runtime to store the virtual address of the library functions used by the binary, such as the the virtual address of the `memcpy` function. Using the OEF of the PLT and of the `apr_pools` objects, we found that each of these objects is shared between a set of threads belonging to a same process. Consequently, we decided to optimize Apache/PHP by pinning all threads belonging to the same Apache processes on the same node. This modification requires less than 10 lines of code and induces a 19.7% performance improvement. This performance improvement is explained by the fact that the optimization drastically reduces the ratio of remote memory accesses. Using this optimization, the Apache/PHP stack performs 10% of its memory accesses on remote memory (75% before optimization).

5.6 Overhead

In this section, we briefly study the overhead and accuracy of MemProf. Note that our observations apply to our three test machines.

The main source of overhead introduced by MemProf is IBS sampling, whose rate is configurable. In order to obtain precise results, we adjust the rate to collect at least 10k samples of instructions accessing DRAM. For most of the applications that we have observed, this translates into generating one IBS instruction every 20k clock cycles, which causes a 5% slowdown. For applications that run for a short period of time or make few DRAM accesses, it may be necessary to increase the sampling rate. In all the applications that we studied, we found it unnecessary to go beyond a sampling rate of one IBS interrupt every 8k cycles, which induces a 20% slowdown.

The costs of IBS processing can be detailed as follows. Discarding a sample (for instructions that do not access DRAM) takes 200 cycles while storing a relevant sample in a per-CPU buffer requires 2k cycles. The storage of samples is currently not heavily optimized. A batch of 10k samples requires 2MB and we preallocate up to 5% of the machine RAM for the buffers, which haven proven acceptable constraints in practice.

The second source of overhead in MemProf is the tracking of the lifecycles of memory objects and threads performed by the user library and the kernel module. The interception of a lifecycle event and its online processing by the user library requires 400 cycles. This tracking introduces a negligible slowdown on the applications that we have studied. The storage of 10k events requires 5.9MB in user buffers, for which we preallocate 20MB of the machine RAM. The processing and storage overheads induced by the kernel-level tracking are significantly lower.

Finally, the offline processing (required to build OEFs and TEFs) for a MemProf trace corresponding to a 1-minute application run takes approximately 5 seconds.

6 Related work

Several projects have focused on profiling for multicore machines. Memphis [12] is a profiler for NUMA-related performance problems. It relies on IBS [5] to find global statically allocated objects that are accessed remotely. VTune [7] uses PEBS (an IBS-like mechanism) to find virtual addresses that are accessed remotely in memory. MemProf extends these works by also identifying the remote memory accesses to dynamically allocated objects, files and code sections and by providing detailed temporal information about these accesses. DProf [14] has been designed to locate kernel structures which induce poor cache behaviors. DProf offers “views" that help developers determine the cause of cache misses. MemProf transposes this work in the context of NUMA-related issues and extends it to application-level profiling. Unlike DProf, MemProf is able to identify object instances, not only object types, a useful feature for many applications. Traditional profilers, such as Oprofile [13] and Perf [19] are able to leverage the performance counters available in recent processors to pinpoint the functions and assembly instructions that perform distant accesses. As seen in Section 5, this information is generally not sufficient to make an accurate NUMA performance diagnosis.

Several research projects have focused on improving NUMA APIs. The Linux kernel [11] offers `cpusets` facilities that can be used to enforce global memory policies on an application (e.g., forcing an application to allocate memory on a limited set of nodes). Goglin et al. [6] have developed a fast `move_pages()` system call in Linux. These works can be leveraged to implement efficient NUMA optimizations, when they are found to be relevant for an application.

Several research projects have used heuristics for automatic mitigation of remote memory accesses. Marathe et al. [10] proposed an automatic page placement tool, which deduces a good placement from a partial run of an application. The tool records memory accesses performed during this partial run and computes the node N from which each allocated memory page P is accessed the most. The application is then re-launched with each page P being allocated from its corresponding node N. This tool is not adapted to applications with multiple execution phases, unlike MemProf. Blagodurov et al. [3] have developed a NUMA-aware scheduler that tries to increase locality and limit contention on critical hardware resources (e.g., memory links and caches). They noticed

that, when a thread is migrated to another node, it is best to also migrate its working set. The number of memory pages to be migrated was determined experimentally. Several works [8, 18] presented automatic page duplication and migration policies. The aggressiveness of a policy can be manually adjusted and has to be carefully chosen to avoid excessive page bouncing between nodes. Thread Clustering [17] is a scheduling technique that tries to improve cache locality by placing threads sharing data on cores sharing a cache. It is not NUMA-aware but, according to its authors, could be easily transformed to improve locality on NUMA machines. Thread Clustering measures the "similarity" of memory accesses performed by distinct threads and clusters threads with high pairwise similarity values. All these works focus on automatically improving the performance of applications and rely on carefully tuned heuristics to increase the locality of data processing. MemProf adopts a complementary approach by offering developers the opportunity to determine why their applications perform remote memory accesses, in order to implement optimizations inside their applications. Besides, MemProf could be used to determine why a given heuristic does not provide the expected performance improvement with a given workload or machine, thus helping the design of heuristics that work on a broader range of situations.

7 Conclusion

Remote memory accesses are a major source or inefficiency on NUMA machines but existing profilers provide little insight on the execution patterns that induce them. We have designed and implemented MemProf, a profiler that outputs precise and configurable views of interactions between threads and memory objects. Our experience with MemProf on four applications shows that it provides programmers with powerful features with respect to the existing tools for performance diagnostics on NUMA multicore machines. We find that MemProf is particularly helpful for applications that exhibit a behavior with at least one of the following characteristics: (i) many object types and/or instances with arbitrary lifespans (and possibly diverse popularities or access patterns), (ii) changing access patterns over time, and (iii) custom memory management routines.

MemProf has two main limitations that we plan to address in our future work. First, it relies on programmers to establish a diagnosis and devise a solution. Second, MemProf is mostly useful for applications that are not cache efficient and that perform a high number of memory accesses. We believe that a more comprehensive profiler should jointly consider the impact of both cache and main memory access patterns, in order to determine the right balance between possibly conflicting optimizations for theses two aspects.

Acknowledgments We thank the anonymous reviewers and our shepherd, Alexandra Fedorova, for their feedback that improved the paper. This work was partially funded by the French SocEDA project and by a hardware grant from INRIA. Some of the experiments were carried out on machines from (i) the MESCAL and MOAIS teams of INRIA/LIG and (ii) the Grid'5000 testbed being developed under the INRIA ALADDIN action with support from CNRS, RENATER, several Universities and other funding bodies (see http://www.grid5000.fr).

References

[1] APACHE HTTP SERVER. `http://httpd.apache.org`.

[2] BIENIA, C., AND LI, K. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation* (2009).

[3] BLAGODUROV, S., ET AL. A case for NUMA-Aware Contention Management on Multicore Systems. In *Proceedings of USENIX ATC* (2011).

[4] BOYD-WICKIZER, S., ET AL. An Analysis of Linux Scalability to Many Cores. In *Proceedings of OSDI* (2010).

[5] DRONGOWSKI, P. J. Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors, 2007. `http://developer.amd.com/Assets/AMD_IBS_paper_EN.pdf`.

[6] GOGLIN, B., AND FURMENTO, N. Enabling High-Performance Memory Migration for Multithreaded Applications on Linux. In *Proceedings of IPDPS* (2009).

[7] INTEL VTUNE. `http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/`.

[8] LAROWE, JR., R. P., ET AL. Evaluation of NUMA Memory Management Through Modeling and Measurements. *IEEE TPDS 3*, 6 (1992), 686–701.

[9] LI, M.-L., ET AL. The ALPBench Benchmark Suite for Complex Multimedia Applications. In *Proceedings of ISWC* (2005).

[10] MARATHE, J., AND MUELLER, F. Hardware Profile-Guided Automatic Page Placement for ccNUMA Systems. In *Proceedings of PPoPP* (2006).

[11] MAUERER, W. *Professional Linux Kernel Architecture*. Wrox Press, 2008.

[12] MCCURDY, C., AND VETTER, J. Memphis: Finding and Fixing NUMA-Related Performance Problems on Multi-Core Platforms. In *Proceedings of ISPASS* (2010).

[13] OPROFILE. A System Profiler for Linux `http://oprofile.sourceforge.net`.

[14] PESTEREV, A., ET AL. Locating Cache Performance Bottlenecks Using Data Profiling. In *Proceedings of EuroSys* (2010).

[15] SPEC. (Standard Performance Evaluation Corporation). SPECweb2005. `http://www.spec.org/web2005/`.

[16] SPRUNT, B. Pentium 4 Performance-Monitoring Features. *Micro, IEEE 22*, 4 (2002), 72–82.

[17] TAM, D., ET AL. Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors. In *Proceedings of EuroSys* (2007).

[18] VERGHESE, B., ET AL. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *Proceedings of ASPLOS* (1996).

[19] WEAVER, V. The Unofficial Linux Perf Events Web-Page. `http://web.eecs.utk.edu/~vweaver1/projects/perf-events/`.

Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications

Jean-Pierre Lozi Florian David Gaël Thomas Julia Lawall Gilles Muller
LIP6/INRIA
firstname.lastname@lip6.fr

Abstract

The scalability of multithreaded applications on current multicore systems is hampered by the performance of lock algorithms, due to the costs of access contention and cache misses. In this paper, we propose a new lock algorithm, Remote Core Locking (RCL), that aims to improve the performance of critical sections in legacy applications on multicore architectures. The idea of RCL is to replace lock acquisitions by optimized remote procedure calls to a dedicated server core. RCL limits the performance collapse observed with other lock algorithms when many threads try to acquire a lock concurrently and removes the need to transfer lock-protected shared data to the core acquiring the lock because such data can typically remain in the server core's cache.

We have developed a profiler that identifies the locks that are the bottlenecks in multithreaded applications and that can thus benefit from RCL, and a reengineering tool that transforms POSIX locks into RCL locks. We have evaluated our approach on 18 applications: Memcached, Berkeley DB, the 9 applications of the SPLASH-2 benchmark suite and the 7 applications of the Phoenix2 benchmark suite. 10 of these applications, including Memcached and Berkeley DB, are unable to scale because of locks, and benefit from RCL. Using RCL locks, we get performance improvements of up to 2.6 times with respect to POSIX locks on Memcached, and up to 14 times with respect to Berkeley DB.

1 Introduction

Over the last twenty years, a number of studies [2, 3, 5, 12, 13, 15, 17, 24, 26, 27] have attempted to optimize the execution of critical sections on multicore architectures, either by reducing access contention or by improving cache locality. Access contention occurs when many threads simultaneously try to enter critical sections that are protected by the same lock, causing the cache line containing the lock to bounce between cores. Cache locality becomes a problem when a critical section accesses shared data that has recently been accessed on another core, resulting in cache misses, which greatly increase the critical section's execution time. Addressing access contention and cache locality together remains a challenge. These issues imply that some applications that work well on a small number of cores do not scale to the number of cores found in today's multicore architectures.

Recently, several approaches have been proposed to execute a succession of critical sections on a single *server* core to improve cache locality [13, 27]. Such approaches also incorporate a fast transfer of control from other *client* cores to the server, to reduce access contention. Suleman *et al.* [27] propose a hardware-based solution, evaluated in simulation, that introduces new instructions to perform the transfer of control, and uses a special fast core to execute critical sections. Hendler *et al.* [13] propose a software-only solution, Flat Combining, in which the server is an ordinary client thread, and the role of server is handed off between clients periodically. This approach, however, slows down the thread playing the role of server, incurs an overhead for the management of the server role, and drastically degrades performance at low contention. Furthermore, neither Suleman *et al.*'s algorithm nor Hender *et al.*'s algorithm can accommodate threads that block within a critical section, which makes them unable to support widely used applications such as Memcached.

In this paper, we propose a new locking technique, Remote Core Locking (RCL), that aims to improve the performance of legacy multithreaded applications on multicore hardware by executing remotely, on one or several dedicated servers, critical sections that access highly contended locks. Our approach is *entirely implemented in software* and targets commodity x86 multicore architectures. At the basis of our approach is the observation that most applications do not scale to the number of cores found in modern multicore architectures, and thus it is possible to *dedicate* the cores that do not contribute to

improving the performance of the application to serving critical sections. Thus, it is not necessary to burden the application threads with the role of server, as done in Flat Combining. RCL also accommodates blocking within critical sections as well as nested critical sections. The design of RCL addresses both access contention and locality. Contention is solved by a fast transfer of control to a server, using a dedicated cache line for each client to achieve busy-wait synchronization with the server core. Locality is improved because shared data is likely to remain in the server core's cache, allowing the server to access such data without incurring cache misses. In this, RCL is similar to Flat Combining, but has a much lower overall overhead.

We propose a methodology along with a set of tools to facilitate the use of RCL in a legacy application. Because RCL serializes critical sections associated with locks managed by the same core, transforming all locks into RCLs on a smaller number of servers could induce false serialization. Therefore, the programmer must first decide which locks should be transformed into RCLs and on which core(s) to run the server(s). For this, we have designed a profiler to identify which locks are frequently used by the application and how much time is spent on locking. Based on this information, we propose a set of simple heuristics to help the programmer decide which locks must be transformed into RCLs. We also have designed an automatic reengineering tool for C programs to transform the code of critical sections so that it can be executed as a remote procedure call on the server core: the code within the critical sections must be extracted as functions and variables referenced or updated by the critical section that are declared by the function containing the critical section code must be sent as arguments, amounting to a context, to the server core. Finally, we have developed a runtime for Linux that is compatible with POSIX threads, and that supports a mixture of RCL and POSIX locks in a single application.

RCL is well-suited to improve the performance of a legacy application in which contended locks are an obstacle to performance, since using RCL enables improving locality and resistance to contention without requiring a deep understanding of the source code. On the other hand, modifying locking schemes to use fine-grained locking or lock-free data structures is complex, requires an overhaul of the source code, and does not improve locality.

We have evaluated the performance of RCL as compared to other locks on a custom latency microbenchmark measuring the execution time of critical sections that access a varying number of shared memory locations. Furthermore, based on the results of our profiler, we have identified Memcached, Berkeley DB with two types of TPC-C transactions, two benchmarks in the SPLASH-2 suite, and three benchmarks in the Phoenix2 suite as applications that could benefit from RCL. In each of these experiments, we compare RCL against the standard POSIX locks and the most efficient approaches for implementing locks of which we are aware: CAS spinlocks, MCS [17] and Flat Combining [13]. Comparisons are made for a same number of cores, which means that there are fewer application threads in the RCL case, since one or more cores are dedicated to RCL servers.

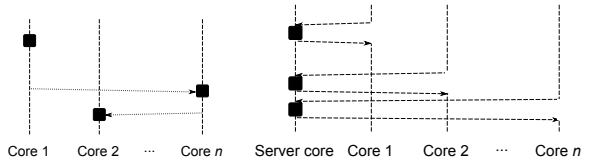
On an Opteron 6172 48-core machine running a 3.0.0 Linux kernel with glibc 2.13, our main results are:

- On our latency microbenchmark, under high contention, RCL is faster than all the other tested approaches, over 3 times faster than the second best approach, Flat Combining, and 4.4 faster than POSIX.
- On our benchmarks, we found that contexts are small, and thus the need to pass a context to the server has only a marginal performance impact.
- On most of our benchmarks, only one lock is frequently used and therefore only one RCL server is needed. The only exception is Berkeley DB which requires two RCL servers to prevent false serialisation.
- On Memcached, for Set requests, RCL provides a speedup of up to 2.6 times over POSIX locks, 2.0 times over MCS and 1.9 times over spinlocks.
- For TPC-C Stock Level transactions, RCL provides a speedup of up to 14 times over the original Berkeley DB locks for 40 simultaneous clients and outperforms all other locks for more than 10 clients. Overall, RCL resists better when the number of simultaneous clients increases.

The rest of the paper is structured as follows. Sec. 2 presents RCL and the use of profiling to automate the reengineering of a legacy application for use with RCL. Sec. 3 describes the RCL runtime. Sec. 4 presents the results of our evaluation. Sec. 5 presents other work that targets improving locking on multicore architectures. Finally, Sec. 6 concludes.

2 RCL Overview

The key idea of RCL is to transfer the execution of a critical section to a server core that is dedicated to one or more locks (Fig. 1). To use this approach, it is necessary to choose the locks for which RCL is expected to be beneficial and to reengineer the critical sections associated with these locks as remote procedure calls. In this section, we first describe the core RCL algorithm, then present a profiling tool to help the developer choose which locks



(a) POSIX locks (b) RCL

Fig. 1: Critical sections with POSIX locks vs. RCL.

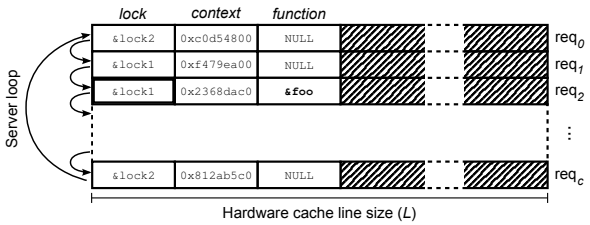


Fig. 2: The request array. Client c_2 has requested execution of the critical section implemented by `foo`.

to implement using RCL and a reengineering tool that rewrites the associated critical sections.

2.1 Core algorithm

Using RCL, a critical section is replaced by a remote procedure call to a procedure that executes the code of the critical section. To implement the remote procedure call, the clients and the server communicate through an array of request structures, specific to each server core (Fig. 2). This array has size $C \cdot L$ bytes, where C is a constant representing the maximum number of allowed clients (a large number, typically much higher than the number of cores), and L is the size of the hardware cache line. Each request structure req_i is L bytes and allows communication between a specific client i and the server. The array is aligned so that each structure req_i is mapped to a single cache line.

The first three machine words of each request req_i contain respectively: (i) the address of the lock associated with the critical section, (ii) the address of a structure encapsulating the *context*, i.e., the variables referenced or updated by the critical section that are declared by the function containing the critical section code, and (iii) the address of a function that encapsulates the critical section for which the client c_i has requested the execution, or `NULL` if no critical section is requested.

Client side To execute a critical section, a client c_i first writes the address of the lock into the first word of the structure req_i, then writes the address of the context structure into the second word, and finally writes the address of the function that encapsulates the code of the critical section into the third word. The client then actively waits for the third word of req_i to be reset to `NULL`, indicating that the server has executed the critical section. In order to improve energy efficiency, if there are less clients than the number of cores available, the SSE3 `monitor/mwait` instructions can be used to avoid spinning: the client will sleep and be woken up automatically when the server writes into the third word of req_i.

Server side A servicing thread iterates over the requests, waiting for one of the requests to contain a non-`NULL` value in its third word. When such a value is found, the servicing thread checks if the requested lock is free and, if so, acquires the lock and executes the critical section using the function pointer and the context. When the servicing thread is done executing the critical section, it resets the third word to `NULL`, and resumes the iteration.

2.2 Profiling

To help the user decide which locks to transform into RCLs, we have designed a profiler that is implemented as a dynamically loaded library and intercepts calls involving POSIX locks, condition variables, and threads. The profiler returns information about the overall percentage of time spent in critical sections, as well as about the percentage of time spent in critical sections for each lock. We define the time spent in a critical section as the total time to acquire the lock (blocking time included), execute the critical section itself, and release the lock. It is measured by reading the cycle counter before and after each critical section, and by comparing the total measured time in critical sections with the total execution time, for each thread. The overall percentage of time spent in critical sections can help identify applications for which using RCL may be beneficial, and the percentage of time spent in critical sections for each lock helps guide the choice of which locks to transform into RCLs. For each lock, the profiler also produces information about the number of cache misses in its critical sections, as these may be reduced by the improved locality of RCL.

Fig. 3 shows the profiling results for 18 applications, including Memcached v1.4.6 (an in-memory cache server), Berkeley DB v5.2.28 (a general-purpose database), the 9 applications of the SPLASH-2 benchmark suite (parallel scientific applications), and the 7 applications of the Phoenix v2.0.0 benchmark suite (MapReduce-based applications) with the "medium" dataset.[1] Raytrace and Memcached are each tested with two different standard working sets, and Berkeley DB is tested with the 5 standard transaction types from TPC-C. A gray box indicates

[1] More information about these applications can be found at the following URLs: `http://memcached.org` (Memcached), `http://www.oracle.com/technetwork/database/berkeleydb` (Berkeley DB), `http://www.capsl.udel.edu/splash` (SPLASH-2) and `http://mapreduce.stanford.edu` (Phoenix2).

that the application has not been run for the number of cores because even at 48 cores, locking is not a problem.

Ten of the tests spend more than 20% of their time in critical sections and thus are candidates for RCL. Indeed, for these tests, the percentage of time spent in critical sections directly depends on the number of cores, indicating that the POSIX locks are one of the main bottlenecks of these applications. We see in Sec. 4.1 that if the percentage of time executing critical sections for a given lock is over 20%, then an RCL will perform better than a POSIX lock, and if it is over 70%, then an RCL will perform better than all other known lock algorithms. We also observe that the critical sections of Memcached/Set incur many cache misses. Finally, Berkeley DB uses hybrid Test-And-Set/POSIX locks, which causes the profiler to underestimate the time spent in critical sections.

2.3 Reengineering legacy applications

If the results of the profiling show that some locks used by the application can benefit from RCL, the developer must reengineer all critical sections that may be protected by the selected locks as a separate function that can be passed to the lock server. This reengineering amounts to an "Extract Method" refactoring [10]. We have implemented this reengineering using the program transformation tool Coccinelle [21], in 2115 lines of code. It has a negligible impact on performance.

The main problem in extracting a critical section into a separate function is to bind the variables needed by the critical section code. The extracted function must receive the values of variables that are initialized prior to the critical section and read within the critical section, and return the values of variables that are updated in the critical section and read afterwards. Only variables local to the function are concerned; alias analysis is not required because aliases involve addresses that can be referenced from the server. The values are passed to and from the server in an auxiliary structure, or directly in the client's cache line in the request array (Fig. 2) if only one value is required. The reengineering also addresses a common case of fine-grained locking, illustrated in lines 5-9 of Fig. 4, where a conditional in the critical section releases the lock and returns from the function. In this case, the code is transformed such that the critical section returns a flag value indicating which unlock ends the critical section, and then the code following the remote procedure call executes the code following the unlock that is indicated by the flag value.

Fig. 5 shows the complete result of transforming the code of Fig. 4. The transformation furthermore modifies various other lock manipulation functions to use the RCL runtime. In particular, the function for initializing a lock receives additional arguments indicating whether the lock should be implemented as an RCL. Finally, the reengineering tool also generates a header file, incorporating the profiling information, that the developer can edit to indicate which lock initializations should create POSIX locks and which ones should use RCLs.

```
1  INT GetJob(RAYJOB *job, INT pid) {
2    ...
3    ALOCK(gm->wplock, pid) /* lock acquisition */
4    wpentry = gm->workpool[pid][0];
5    if (!wpentry) {
6      gm->wpstat[pid][0] = WPS_EMPTY;
7      AULOCK(gm->wplock, pid) /* lock release */
8      return (WPS_EMPTY);
9    }
10   gm->workpool[pid][0] = wpentry->next;
11   AULOCK(gm->wplock, pid) /* lock release */
12   ...
13 }
```

Fig. 4: Critical section from `raytrace/workpool.c`.

```
1  union instance {
2    struct input { INT pid; } input;
3    struct output { WPJOB *wpentry; } output;
4  };
5
6  void function(void *ctx) {
7    struct output *outcontext = &(((union instance *)ctx)->output);
8    struct input  *incontext  = &(((union instance *)ctx)->input);
9    WPJOB *wpentry; INT pid=incontext->pid;
10   int ret=0;
11   /* start of original critical section code */
12   wpentry = gm->workpool[pid][0];
13   if (!wpentry) {
14     gm->wpstat[pid][0] = WPS_EMPTY;
15     /* end of original critical section code */
16     ret = 1;
17     goto done;
18   }
19   gm->workpool[pid][0] = wpentry->next;
20   /* end of original critical section code */
21 done:
22   outcontext->wpentry = wpentry;
23   return (void *)(uintptr_t)ret;
24 }
25
26 INT GetJob(RAYJOB *job, INT pid) {
27   int ret;
28   union instance instance = { pid, };
29   ...
30   ret = liblock_exec(&gm->wplock[pid], &instance, &function);
31   wpentry = instance.output.wpentry;
32   if (ret) { if (ret == 1) return (WPS_EMPTY); }
33   ...
34 }
```

Fig. 5: Transformed critical section.

3 Implementation of the RCL Runtime

Legacy applications may use ad-hoc synchronization mechanisms and rely on libraries that themselves may block or spin. The core algorithm, of Sec. 2.1, only refers to a single servicing thread, and thus requires that this thread is never blocked at the OS level and never spins in an active waitloop. In this section, we describe how the RCL runtime ensures liveness and responsiveness in these cases, and present implementation details.

Application		% in CS = f(# cores)						Lock usage for # max. core			
		1	4	8	16	32	48	Description	#	# L2 cache misses/CS	% in CS
SPLASH-2	Radiosity	4.3%	8.8%	15.6%	43%	79.3%	84.5%	Linked list access	1	1.6	82.0 %
	Raytrace Balls4	0.5%	1.3%	1.9%	3.3%	17%	40.1%	Counter increment	1	1.3	32.32 %
	Raytrace Car	12.2%	25.9%	51.4%	71.9%	85.5%	87.6%	Counter increment	1	0.6	79.95 %
	Barnes						3.1%				
	FMM						5.0%				
	Ocean Cont.					0.3%†					
	Ocean Non Cont.					0.3%†					
	Volrend						6.8%				
	Water-nsquared						3.6%				
	Water-spatial						0.5%				
Phoenix 2	Linear Regression	0.9%	25.2%	43.7%	66.9%	60.8%	83.6%	Task queue access	1	4.0	83.6%
	String Match	0.1%	4.7%	11.7%	24.2%	35.0%	63.4%	Task queue access	1	3.9	63.4%
	Matrix Multiply	0.9%	26.2%	45.9%	64.8%	79.2%	92.7%	Task queue access	1	3.4	92.7%
	Histogram						12.7%				
	PCA						11.6%				
	KMeans						1.5%				
	Word Count						3.2%				
Memcached	Get	6.7%	30.2%	50.1%	76.3%	22 cores: 84.9%‡		Hashtable access	1	3.6	84.7%
	Set	4.8%	28.7%	44.6%	54.4%	22 cores: 55.4%‡		Hashtable access	1	32.7	55.3%
Berkeley DB with TPC-C	Payment						5.80%				
	New Order						1.55%				
	Order Status	0.8%	0.8%	0.3%	2.0%	35.8%	52.9%	DB struct. access	11	4.2	52.9%
	Delivery						1.58%				
	Stock Level	0.0%	0.4%	0.2%	2.2%	0.4%	55.5%	DB struct. access	11	3.2	55.5%

† Number of cores must be a power of 2. ‡ Other cores are executing clients.

Fig. 3: Profiling results for the evaluated applications.

3.1 Ensuring liveness and responsiveness

Three sorts of situations may induce liveness or responsiveness problems. First, the servicing thread could be blocked at the OS level, e.g., because a critical section tries to acquire a POSIX lock that is already held, performs an I/O, or waits on a condition variable. Indeed, we have found that roughly half of the multithreaded applications that use POSIX locks in Debian 6.0.3 (October 2011) also use condition variables. Second, the servicing thread could spin if the critical section tries to acquire a spinlock or a nested RCL, or implements some form of ad hoc synchronization [29]. Finally, a thread could be preempted at the OS level when its timeslice expires [20], or because of a page fault. Blocking and waiting within a critical section risk deadlock, because the server is unable to execute critical sections associated with other locks, even when doing so may be necessary to allow the blocked critical section to unblock. Additionally, blocking, of any form, including waiting and preemption, degrades the responsiveness of the server because a blocked thread is unable to serve other locks managed by the same server.

Ensuring liveness To ensure liveness, the RCL runtime manages a pool of threads on each server such that when a servicing thread blocks or waits there is always at least one other *free* servicing thread that is not currently executing a critical section and this servicing thread will eventually be elected. To ensure the existence of a free servicing thread, the RCL runtime provides a *management thread*, which is activated regularly at each expiration of a *timeout* (we use the Linux time-slice value) and runs at highest priority. When activated, the management thread checks that at least one of the servicing threads has made progress since its last activation, using a server-global flag *is_alive* that is set by the servicing threads. If it finds that this flag is still cleared when it wakes up, it assumes that all servicing threads are either blocked or waiting and adds a free thread to the pool of servicing threads.

Ensuring responsiveness The RCL runtime implements a number of strategies to improve responsiveness. First, it avoids thread preemption from the OS scheduler by using the POSIX FIFO scheduling policy, which allows a thread to execute until it blocks or yields the processor. Second, it tries to reduce the delay before an unblocked servicing thread is rescheduled by minimizing the number of servicing threads, thus minimizing the length of the FIFO queue. Accordingly, a servicing thread suspends when it observes that there is at least one other free servicing thread, i.e., one other thread able to handle requests. Third, to address the case where all servicing threads are blocked in the OS, the RCL runtime uses a *backup thread*, which has lower priority than all servicing threads, that simply clears the *is_alive* flag and wakes up the management thread. Finally, when a critical section needs to execute a nested RCL managed by the same core and the lock is already owned by another servicing thread, the servicing thread immediately yields, to allow the owner of the lock to release it.

The use of the FIFO policy raises two further liveness issues. First, FIFO scheduling may induce a priority inversion between the backup thread and the servicing threads or between the servicing threads and the management

thread. To avoid this problem, the RCL runtime uses only lock-free algorithms and threads never wait on a resource. Second, if a servicing thread is in an active wait loop, it will not be preempted, and a free thread will not be elected. When the management thread detects no progress, i.e., *is_alive* is false, it thus also acts as a scheduler, electing a servicing thread by first decrementing and then incrementing the priorities of all the servicing threads, effectively moving them to the end of the FIFO queue. This is expensive, but is only needed when a thread spins for a long time, which is a sign of poor programming, and is not triggered in our benchmarks.

3.2 Algorithm details

We now describe in detail some issues of the algorithm.

Serving RCLs Alg. 1 shows the code executed by a servicing thread. The *fast path* (lines 6-16) is the only code that is executed when there is only one servicing thread in the pool. A *slow path* (lines 17-24), is additionally executed when there are multiple servicing threads.

Lines 9-15 of the fast path implement the RCL server loop as described in Sec. 2.1 and indicates that the servicing thread is not free by decrementing (line 8) and incrementing (line 16) *number_of_free_threads*. Because the thread may be preempted due to a page fault, all operations on variables shared between the threads, including *number_of_free_threads*, must be atomic.[2] To avoid the need to reallocate the request array when new client threads are created, the size of the request array is fixed and chosen to be very large (256K), and the client identifier allocator implements an adaptive long-lived renaming algorithm [6] that keeps track of the highest client identifier and tries to reallocate smaller ones.

The slow path is executed if the active servicing thread detects that other servicing threads exist (line 17). If the other servicing threads are all executing critical sections (line 18), the servicing thread simply yields the processor (line 19). Otherwise, it goes to sleep (lines 21-24).

Executing a critical section A client that tries to acquire an RCL or a servicing thread that tries to acquire an RCL managed by another core submits a request and waits for the function pointer to be cleared (Alg. 2, lines 6-9). If the RCL is managed by the same core, the servicing thread must actively wait until the lock is free. During this time it repetitively yields, to give the CPU to the thread that owns the lock (lines 11-12).

Management and backup threads If, on wake up, the management thread observes, based on the value of *is_alive*, that none of the servicing threads has progressed since the previous timeout, then it ensures that at least one free thread exists (Alg. 3, lines 8-19) and forces the election (lines 20-27) of a thread that has not been recently elected. The backup thread (lines 31-34) simply sets *is_alive* to false and wakes up the management thread.

[2] Since a server's atomic operations are core-local, we have implemented optimized atomic CAS and increment operations without using the costly x86 instruction prefix `lock` that cleans up the write buffers.

Algorithm 1: Structures and server thread

```
1  structures:
       lock_t:      { server_t* server, bool is_locked };
       request_t:   { function_t* code, void* context, lock_t* lock };
       thread_t:    { server_t* server, int timestamp, bool is_servicing };
       server_t:    { List<thread_t*> all_threads,
                      LockFreeStack<thread_t*> prepared_threads,
                      int number_of_free_threads, number_of_servicing_threads,
                      int timestamp, boolean is_alive, request_t* requests }
2  global variables:  int number_of_clients;
3  function rcl_servicing_thread(thread_t* t)
4      var server_t* s := t->server;
5      while true do
6          s->is_alive := true;
7          t->timestamp := s->timestamp;
           // This thread is not free anymore.
8          local_atomic(s->number_of_free_threads--);
9          for i := 0 to number_of_clients do
10             r := s->requests[i];
11             if r->code ≠ null and
12                 local_CAS(&r->lock->is_locked, false, true) = false
                   then
                       // Execute the critical section
13                     r->code(r->context);
                       // Indicate client execution completion
14                     r->code := null;
15                     r->lock->is_locked := false;
           // This thread is now free
16         local_atomic(s->number_of_free_threads++);
           // More than one servicing thread means blocked threads exist
17         if s->number_of_servicing_threads > 1 then
18             if s->number_of_free_threads <= 1 then
19                 yield(); // Allow other busy servicing threads to run
20             else
                   // Keep only one free servicing thread
21                 t->is_servicing := false;
22                 local_atomic(s->number_of_servicing_threads--);
23                 local_atomic_insert(s->prepared_threads, t);
                   // Must be atomic because the manager could wake
                      up the thread before the sleep (use futex).
24                 atomic(if not t->is_servicing then sleep)
```

4 Evaluation

We first present a microbenchmark that identifies when critical sections execute faster with RCL than with the other lock algorithms. We then correlate this information with the results of the profiler, so that a developer can use the profiler to identify which locks to transform into RCLs. Finally, we analyze the performance of the applications identified by the profiler for all lock algorithms. Our evaluations are performed on a 48-core machine having four 12-core Opteron 6172 processors, running Ubuntu 11.10 (Linux 3.0.0), with gcc 4.6.1 and glibc 2.13.

Algorithm 2: Executing a critical section

```
1  thread local variables:
2      int th_client_index; bool is_server_thread; server_t* my_server;
3  function execute_cs(lock_t* lock, function_t* code, void* context)
4      var request_t * r := &lock->server->requests[th_client_index];
5      if ¬is_server_thread or my_server ≠ lock->server then
           // RCL to a remote core
6          r->lock := lock; r->context := context; r->code := code;
7          while r->code ≠ null do
8              ;
9          return;
10     else
           // Local nested lock, wait until the lock is free
11         while local_CAS(&lock->is_locked,false,true) = true do
               // Another thread on the server owns the lock
12             yield();
           // Execute the critical section
13         code(context);
14         lock->is_locked := false;
15         return;
```

Algorithm 3: Management

```
1  function management_thread(server_t *s)
2      var thread_t* t;
3      s->is_alive := false;
4      s->timestamp := 1;
5      while true do
6          if s->is_alive = false then
7              s->is_alive := true;
               // Ensure that a thread can handle remote requests
8              if s->number_of_free_threads = 0 then
                   // Activate a prepared thread or create a new thread
9                  local_atomic(s->number_of_servicing_threads++);
10                 local_atomic(s->number_of_free_threads++);
11                 t := local_atomic_remove(s->prepared_threads);
12                 if t = null then
13                     t := allocate_thread(s);
14                     insert(s->all_threads,t);
15                     t->is_servicing := true;
16                     t.start(prio_servicing);
17                 else
18                     t->is_servicing := true;
19                     wakeup(t);
               // Elect a thread that has not recently been elected
20             while true do
21                 for t in s->all_threads do
22                     if t->is_servicing = true
23                         and t->timestamp < s->timestamp then
24                         t->timestamp = s->timestamp;
25                         elect(t);
26                         goto end;
                   // All threads were elected once, begin a new cycle
27                 s->timestamp++
28         else
29             s->is_alive := false;
30         sleep(timeout);
31 function backup_thread(server_t *s)
32     while true do
33         s->is_alive := false;
34         wakeup the management thread;
```

4.1 Comparison with other locks

We have developed a custom microbenchmark to measure the performance of RCL relative to four other lock algorithms: CAS Spinlock, POSIX, MCS [18] and Flat Combining [13]. These algorithms are briefly presented in Fig. 6. To our knowledge, MCS and Flat Combining are currently the most efficient.

Spinlock	CAS loop on a shared cache line.
POSIX	CAS, then sleep.
MCS	CAS to insert the pending CS at the end of a shared queue. Busy wait for completion of the previous CS on the queue.
Flat Combining	Periodic CAS to elect a client that acts as a server, periodic collection of unused requests. Provides a generic interface, but not combining, as appropriate to support legacy applications: server only iterates over the list of pending requests.

Fig. 6: The evaluated lock algorithms.

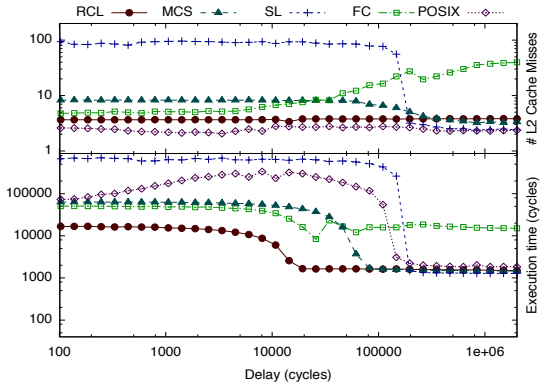
Our microbenchmark executes critical sections repeatedly on all cores, except one that manages the lifecycle of the threads. For RCL, this core also executes the RCL server. We vary the *degree of contention* on the lock by varying the delay between the execution of the critical sections: the shorter the delay, the higher the contention. We also vary the *locality* of the critical sections by varying the number of shared cache lines each one accesses (references and updates). To ensure that cache line accesses are not pipelined, we construct the address of the next memory access from the previously read value [30].

Fig. 7(a) presents the average number of L2 cache misses (top) and the average execution time of a critical section (bottom) over 5000 iterations when critical sections access one shared cache line. This experiment measures the effect of lock access contention. Fig. 7(b) then presents the increase in execution time incurred when each critical section instead accesses 5 cache lines. This experiment measures the effect of data locality of shared cache lines. Highlights are summarized in Fig.7(c).

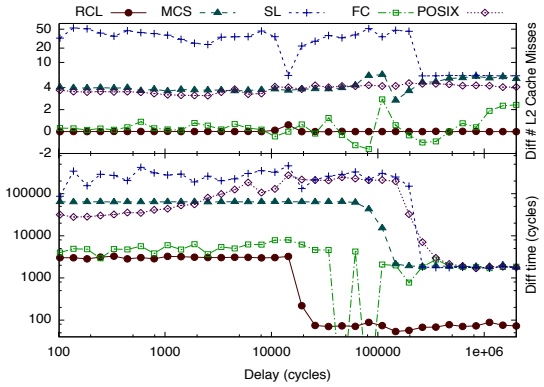
With one shared cache line, at high contention, RCL has a better execution time than all other lock algorithms, with an improvement of at least 3 times.[3] This improvement is mainly due to the absence of CAS in the lock implementation. Flat Combining is second best, but RCL performs better because Flat Combining has to periodically elect a new combiner. At low contention, RCL is slower than Spinlock by only 209 cycles. This is negligible since the lock is seldom used. In this case, Flat Combining is not efficient because after executing its critical section, the combiner must iterate over the list of requests before resuming its own work.

RCL incurs the same number of cache misses when each critical section accesses 5 cache lines as it does for

[3] Using the SSE3 `monitor/mwait` instructions on the client side when waiting for a reply from the server, as described in Sec. 2.1, induces a latency overhead of less than than 30% at both high and low contention. This makes the energy-efficient version of RCL quantitatively similar to the original RCL implementation presented here.



(a) Execution with one shared cache line per CS



(b) Difference between one and five cache lines per CS

	High contention (10^2 cycles)					Low contention (2.10^6 cycles)				
	CAS/CS	Execution time (cycles)		Locality (misses)		CAS/CS	Execution time (cycles)		Locality (misses)	
Spinlock	N	Bad	(672889)	Very Bad	(+53.0 misses)	N	Good	(1288)	Bad	(+5.2)
POSIX	1	Medium	(73024)	Bad	(+3.8 misses)	1	Medium	(1826)	Bad	(+4.0)
MCS	1	Medium	(63553)	Bad	(+4.0 misses)	1	Good	(1457)	Bad	(+4.8)
Flat Combining	ε	Medium	(50447)	Good	(+0.3 misses)	1	Bad	(15060)	Medium	(+2.4)
RCL	0	Good	(16682)	Good	(+0.0 misses)	0	Good	(1494)	Good	(+0.0)

(c) Comparison of the lock algorithms

Fig. 7: Microbenchmark results. Each data point is the average of 30 runs.

one cache line, as the data remains on the RCL server. At low contention, each request is served immediately, and the performance difference is also quite low. At higher contention, each critical section has to wait for the others to complete, incurring an increase in execution time of roughly 47 times the increase at low contention. Like RCL, Flat Combining has few or no extra cache misses at high contention, because cache lines stay with the combiner, which acts as a server. At low contention, the number of extra cache misses is variable, because the combiner often has no other critical sections to execute. These extra cache misses increase the execution time. POSIX and MCS have 4 extra cache misses when reading the 4 extra cache lines, and incur a corresponding execution time increase. Finally, Spinlock is particularly degraded at high contention when accessing 5 cache lines, as the longer duration of the critical section increases the amount of time the thread spins, and thus the number of CAS it executes.

To estimate which locks should be transformed into RCLs, we correlate the percentage of time spent in critical sections observed using the profiler with the critical section execution times observed using the microbenchmark. Fig. 8 shows the result of applying the profiler to the microbenchmark in the one cache line case with POSIX locks.[4] To know when RCL becomes better than all other locks, we focus on POSIX and MCS: Flat Combining is always less efficient than RCL and Spinlock is only efficient at very low contention. We have marked the delays at which, as shown in Fig. 7(a), the critical section execution time begins to be significantly higher when using POSIX and MCS than when using RCL. RCL becomes more efficient than POSIX when 20% of the application time is devoted to critical sections, and it becomes more efficient than MCS when this ratio is 70%. These results are preserved, or improved, as the number of accessed cache lines increases, because the execution time increases more for the other algorithms than for RCL.

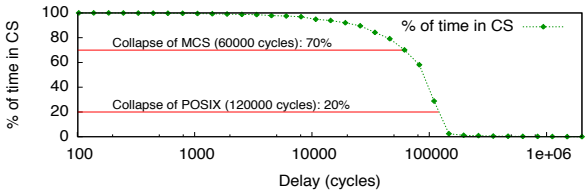
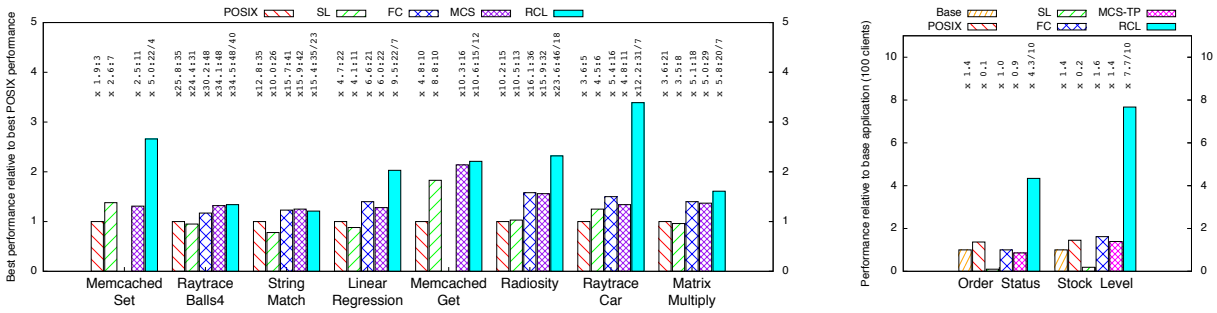


Fig. 8: CS time in the microbenchmark with POSIX locks.

4.2 Application performance

The two metrics offered by the profiler, i.e. the time spent in critical sections and the number of cache misses, do not, of course, completely determine whether an application will benefit from RCL. Many other factors (critical section length, interactions between locks, etc.) affect critical section execution. We find, however, that using the time spent in critical sections as our main metric and the number of cache misses in critical sections as a secondary metric works well; the former is a good indicator

[4] Our analysis assumes that the targeted applications use POSIX locks, but a similar analysis could be made for any type of lock.



(a) Applications with one lock bottleneck

(b) Berkeley DB

Fig. 9: Best performance for each type of lock relative to the best performance for POSIX locks.

of contention, and the latter of data locality.

To evaluate the performance of RCL, we have measured the performance of applications listed in Fig. 3 with the lock algorithms listed in Fig. 7. Memcached with Flat Combining is omitted, because it periodically blocks on condition variables, which Flat Combining does not support. We present only the results for the applications (and locks) that the profiler indicates as potentially interesting. Replacing the other locks has no performance impact.

Fig. 9(a) presents the results for all of the applications for which the profiler identified a single lock as the bottleneck. For RCL, each of these applications uses only one server core. Thus, for RCL, we consider that we use N cores if we have $N-1$ threads and 1 server, while we consider that we use N cores if we have N threads for the other lock algorithms. The top of the figure ($\mathrm{x}\alpha : n/m$) reports the improvement α over the execution time of the original application on one core, the number n of cores that gives the shortest execution time (i.e., the scalability peak), and the minimal number m of cores for which RCL is faster than all other locks. The histograms show the ratio of the shortest execution time for each application using POSIX locks to the shortest execution time with each of the other lock algorithms.[5]

Fig. 9(b) presents the results for Berkeley DB with 100 clients (and hence 100 threads) running TPC-C's Order Status and Stock Level transactions. Since MCS cannot handle more than 48 threads, due to the convoy effect, we have also implemented MCS-TP [12], a variation of MCS with a spinning timeout to resist convoys. In the case of RCL, the two most used locks have been placed on two different RCL servers, leaving 46 cores for the clients. Additionally, we study the impact of the number of simultaneous clients on the number of transactions treated per second for Stock Level transactions (see Fig. 11).

Performance analysis For the applications that spend 20-70% of their time in critical sections when using POSIX locks (Raytrace/Balls4, String Match, and Memcached/Set), RCL gives significantly better performance than POSIX locks, but in most cases it gives about the same performance as MCS and Flat Combining, as predicted by our microbenchmark. For Memcached/Set, however, which spends only 54% of the time in critical sections when using POSIX locks, RCL gives a large improvement over all other approaches, because it significantly improves cache locality. When using POSIX locks, Memcached/Set critical sections have on average 32.7 cache misses, which roughly correspond to accesses to 30 shared cache lines, plus the cache misses incurred for the management of POSIX locks. Using RCL, the 30 shared cache lines remain in the server cache. Fig. 10 shows that for Memcached/Set, RCL performs worse than other locks when fewer than four cores are used due to the fact that one core is lost for the server, but from 5 cores onwards, this effect is compensated by the performance improvement offered by RCL.

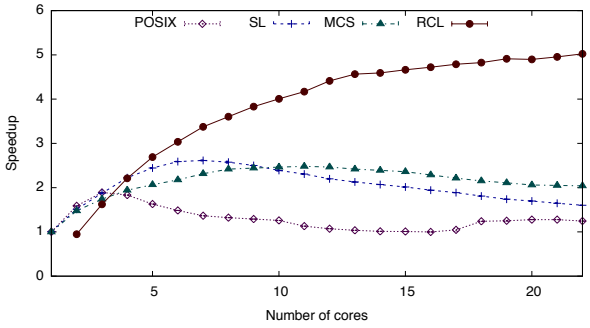


Fig. 10: Memcached/Set speedup.

For most of the applications that spend more than 70% of their time in critical sections when using POSIX locks (Radiosity, Raytrace/Car, and Linear Regression), RCL gives a significant improvement over all the other lock algorithms, again as predicted by our microbenchmark. Matrix Multiply, however, spends over 90% of its time in critical sections when using POSIX locks, but shows

[5]For Memcached, the execution time is the time for processing 10,000 requests.

only a slight performance improvement. This application is intrinsically unable to scale for the considered data set; even though the use of RCL reduces the amount of time spent in critical sections to 1% (Fig. 12), the best resulting speedup is only 5.8 times for 20 cores. Memcached/Get spends more than 80% of its time in critical sections, but is only slightly improved by RCL as compared to MCS. Its critical sections are long and thus acquiring and releasing locks is less of a bottleneck than with other applications.

In the case of Berkeley DB, RCL achieves a speedup of 4.3 for Order Status transactions and 7.7 for Stock Level transactions with respect to the original Berkeley DB implementation for 100 clients. This is better than expected, since, according to our profiler, the percentage of time spent in critical sections is respectively only 53% and 55%, i.e. less than the 70% threshold. This is due to the fact that Berkeley DB uses hybrid Test-And-Set/POSIX locks, and our profiler was designed for POSIX locks: the time spent in the Test-And-Set loop is not included in the "time in critical sections" metric.

When the number of clients increases, the throughput of all implementations degrades. Still, RCL performs better than the other lock algorithms, even though two cores are reserved for the RCL servers and thus do not directly handle requests. In fact, the cost of the two server cores is amortized from 5 clients onwards. The best RCL speedup over the original implementation is for 40 clients with a ratio of 14 times. POSIX is robust for a large number of threads and comes second after RCL. MCS-TP [12] resists convoys but with some overhead. MCS-TP and Flat Combining have comparable performance.

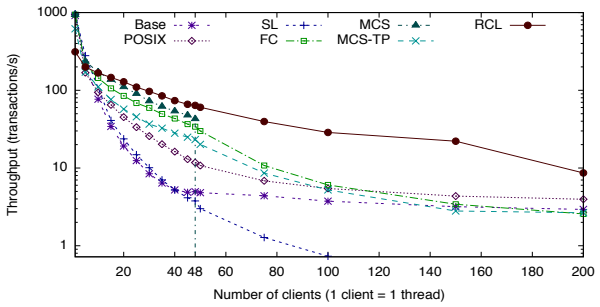


Fig. 11: Berkeley DB/Stock Level throughput.

Locality analysis Figure 12 presents the number of L2 cache misses per critical section observed on the RCL server for the evaluated applications. Critical sections trigger on average fewer than 4 cache misses, of which the communication between the client and the server itself costs one cache miss. Thus, on average, at most 3 cache lines of context information are accessed per critical section. This shows that passing variables to and from the server does not hurt performance in the evaluated applications.

Application	L2 cache misses on the RCL server
Radiosity	2.2
Raytrace/Car	1.8
Raytrace/Balls4	1.8
Linear Regression	2.4
Matrix Multiply	3.2
String Match	3.2
Memcached/Get	N/A†
Memcached/Set	N/A†
Berkeley DB/Order Status	3.3
Berkeley DB/Stock Level	3.6

† We are currently unable to collect L2 cache misses when using blocking on RCL servers.

Fig. 12: Number of L2 cache misses per CS.

False Serialization A difficulty in transforming Berkeley DB for use with RCL is that the call in the source code that allocates the two most used locks also allocates nine other less used locks. The RCL runtime requires that for a given lock allocation site, all allocated locks are implemented in the same way, and thus all 11 locks must be implemented as RCLs. If all 11 locks are on the same server, their critical sections are artificially serialized. To prevent this, the RCL runtime makes it possible to choose the server core where each lock will be dispatched.

To study the impact of this false serialization, we consider two metrics: *false serialization rate* and *use rate*. The false serialization rate is the ratio of the number of iterations over the request array where the server finds critical sections associated with at least two different locks to the number of iterations where at least one critical section is executed.[6] The use rate measures the server workload. It is computed as the total number of executed critical sections divided by the number of iterations where at least one critical section is executed, giving the average number of clients waiting for a critical section on each iteration, which is then divided by the number of cores. Therefore, a use rate of 1.0 means that all elements of the array contain pending critical section requests, whereas a low use rate means that the server mostly spins on the request array, waiting for critical sections to execute.

Fig. 13 shows the false serialization rate and the use rate for Berkeley DB (100 clients, Stock Level): (i) with one server for all locks, and (ii) with two different servers for the two most used locks as previously described. Using one server, the false serialization rate is high and has a significant impact because the use rate is also high. When using two servers, the use rate of the two servers goes down to 5% which means that they are no longer saturated and that false serialization is eliminated. This allows us to improve the throughput by 50%.

[6] We do not divide by the total number of iterations, because there are many iterations in application startup and shutdown that execute no critical sections and have no impact on the overall performance.

	False serialization rate	Use rate	Transactions/s
One server	91%	81%	18.9
Two servers	<1% / <1%	5%/5%	28.7

Fig. 13: Impact of false serialization with RCL.

5 Related Work

Many approaches have been proposed to improve locking [1, 8, 13, 15, 24, 26, 27]. Some improve the fairness of lock algorithms or reduce the data bus load [8, 24]. Others switch automatically between blocking locks and spinlocks depending on the contention rate [15]. Others, like RCL, address data locality [13, 27].

GLocks [1] addresses at the hardware level the problem of latency due to cache misses of highly-contended locks by using a token-ring between cores. When a core receives the token, it serves a pending critical section, if it has one, and then forwards the token. However, only one token is used, so only one lock can be implemented. Suleman *et al.* [27] transform critical sections into remote procedure calls to a powerful server core on an asymmetric multicore. Their communication protocol is also implemented in hardware and requires a modified processor. They do not address blocking within critical sections, which can be a problem with legacy library code. RCL works on legacy hardware and allows blocking.

Flat Combining [13], temporarily transforms the owner of a lock into a server for other critical sections. Flat Combining is unable to handle blocking in a critical section, because there is only one combiner. At low contention, Flat Combining is not efficient because the combiner has to check whether pending requests exist, in addition to executing its own code. In RCL, the server may also uselessly scan the array of pending requests, but as the server has no other code to execute, it does not incur any overall delay. Sridharan *et al.* [26] increase data locality by associating an affinity between a core and a lock. The affinity is determined by intercepting Futex [9] operations and the Linux scheduler is modified so as to schedule the lock requester to the preferred core of the lock. This technique does not address the access contention that occurs when several cores try to enter their critical sections.

Roy *et al.* [23] have proposed a profiling tool to identify critical sections that work on disjoint data sets, in order to optimize them by increasing parallelism. This approach is complementary to ours. Lock-free structures have been proposed in order to avoid using locks for traditional data structures such as counters, linked lists, stacks, or hashtables [14, 16, 25]. These approaches never block threads. However, such techniques are only applicable to the specific types of data structures considered. For this reason, locks are still commonly used on multicore architectures. Finally, experimental operating systems and databases designed with manycore architectures in mind use data replication to improve locality [28] and even RPC-like mechanisms to access shared data from remote cores [4, 7, 11, 19, 22]. These solutions, however, require a complete overhaul of the operating system or database design. RCL, on the other hand, can be used with current systems and applications with few modifications.

6 Conclusion

RCL is a novel locking technique that focuses on both reducing lock acquisition time and improving the execution speed of critical sections through increased data locality. The key idea is to migrate critical-section execution to a server core. We have implemented an RCL runtime for Linux that supports a mixture of RCL and POSIX locks in a single application. To ease the reengineering of legacy applications, we have designed a profiling-based methodology for detecting highly contended locks and implemented a tool that transforms critical sections into remote procedure calls. Our performance evaluations on legacy benchmarks and widely used legacy applications show that RCL improves performance when an application relies on highly contended locks.

In future work, we will consider the design and implementation of an adaptive RCL runtime. Our first goal will be to be able to dynamically switch between locking strategies, so as to dedicate a server core only when a lock is contented. Second, we want to be able to migrate locks between multiple servers, to dynamically balance the load and avoid false serialization. One of the challenges will be to implement low-overhead run-time profiling and migration strategies. Finally, we will explore the possibilities of RCL for designing new applications.

Availability The implementation of RCL as well as our test scripts and results are available at `http://rclrepository.gforge.inria.fr`.

Acknowledgments We would like to thank Alexandra Fedorova and our shepherd Wolfgang Schröder-Preikschat for their insightful comments and suggestions.

References

[1] J. L. Abellán, J. Fernández, and M. E. Acacio. GLocks: Efficient support for highly-contended locks in many-core CMPs. In *25th IPDPS*, 2011.

[2] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *ISCA'89*, pages 396–406, 1989.

[3] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization for Java. In *PLDI'98*, pages 258–268, 1998.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schuepbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP'09*, pages 29–44, 2009.

[5] L. Boguslavsky, K. Harzallah, A. Kreinen, K. Sevcik, and A. Vainshtein. Optimal strategies for spinning and blocking. *Journal of Parallel and Distributed Computing*, 1993.

[6] A. Brodsky, F. Ellen, and P. Woelfel. Fully-adaptive algorithms for long-lived renaming. In *DISC '06*, pages 413–427, 2006.

[7] E. M. Chaves Jr., P. Das, T. J. LeBlanc, B. D. Marsh, and M. L. Scott. Kernel-kernel communication in a shared-memory multiprocessor. *Concurrency - Practice and Experience*, 5(3):171–191, 1993.

[8] T. S. Craig. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, Department of Computer Science, University of Washington, Feb. 2003.

[9] U. Drepper and I. Molnar. Native POSIX thread library for Linux. Technical report, RedHat, 2003.

[10] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[11] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *OSDI'99*, pages 87–100.

[12] B. He, W. N. Scherer III, and M. L. Scott. Preemption adaptivity in time-published queue-based spin locks. In *11th International Conference on High Performance Computing*, 2005.

[13] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA'10*, pages 355–364, 2010.

[14] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Elsevier, 2008.

[15] F. R. Johnson, R. Stoica, A. Ailamaki, and T. C. Mowry. Decoupling contention management from scheduling. In *ASPLOS'10*, pages 117–128, 2010.

[16] E. Ladan-Mozes and N. Shavit. An optimistic approach to lock-free FIFO queues. *Distributed Computing*, 20(5):323–341, 2008.

[17] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM TOCS*, 9(1):21–65, 1991.

[18] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *ASPLOS*, pages 269–278. ACM, 1991.

[19] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *SOSP'09*, pages 221–234, 2009.

[20] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *ICDCS '82*, pages 22–30, 1982.

[21] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys*, pages 247–260, 2008.

[22] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *Proceedings of the VLDB Endowment (PVLDB)*, 3(1):928–939, 2010.

[23] A. Roy, S. Hand, and T. Harris. Exploring the limits of disjoint access parallelism. In *HotPar'09*, pages 8–8, 2009.

[24] M. L. Scott and W. N. Scherer. Scalable queue-based spin locks with timeout. In *PPoPP'01*, pages 44–52, 2001.

[25] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *JACM*, 53(3):379–405, May 2006.

[26] S. Sridharan, B. Keck, R. Murphy, S. Chandra, and P. Kogge. Thread migration to improve synchronization performance. In *In Workshop on Operating System Interference in High Performance Applications*, 2006.

[27] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS*, pages 253–264, 2009.

[28] S. B. Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. In *OSDI'08*.

[29] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *OSDI'10*, pages 1–8, 2010.

[30] K. Yotov, K. Pingali, and P. Stodghill. Automatic measurement of memory hierarchy parameters. In *SIGMETRICS '05*, pages 181–192, 2005.

The Click2NetFPGA Toolchain

Teemu Rinta-aho
NomadicLab
Ericsson Research
Jorvas, Finland

Mika Karlstedt
NomadicLab
Ericsson Research
Jorvas, Finland

Madhav P. Desai
Department of Electrical Engineering
Indian Institute of Technology (Bombay)
Mumbai, India

Abstract

High Level Synthesis (HLS) is a promising technology where algorithms described in high level languages are automatically transformed into a hardware design. Although many HLS tools exist, they are mainly targeting developers who want to use a high level programming language to design hardware modules. They are not designed to automatically compile a complete software system, such as a network packet processing application, into a hardware design.

In this paper, we describe a compiler toolchain that automatically transforms existing software in a limited domain to a functional hardware design. We have selected the Click Modular Router as the input system, and the Stanford NetFPGA as the target hardware platform. Our toolchain uses LLVM to transform Click C++ code into a form suitable for hardware implementation and then uses AHIR, a high level synthesis toolchain, to produce a VHDL netlist.

The resulting netlist has been verified with actual hardware on the NetFPGA platform. The resulting hardware can achieve 20-50 % of the performance compared to version handwritten in Verilog. We expect that improvements on the toolchain could provide better performance, but for the first prototype the results are good. We feel that one of the biggest contribution of this work is that it shows some new principles of high-level synthesis that could also be applied to different domains, source languages and targets.

1 Introduction

Writing packet processing applications in software offers a flexible and easy way for experimentation and product development. Hardware-based implementations, on the other hand, characterised by parallel operations and inflexibility, result in better energy efficiency and higher operational speed. At the same time producing hardware is orders of magnitude more expensive in terms of both design and manufacturing than producing software.

Click modular router [12] is a popular tool for writing software routers. Stanford NetFPGA [14] is a similarly flexible platform for developing hardware routers. While programming packet processing applications in Click C++ is easy and flexible, describing even a simple application in VHDL/Verilog (for the NetFPGA) is a relatively major undertaking. Combining the good sides from both of the two could be an enabler for rapid implementation of efficient packet processing applications.

An ideal software-to-hardware toolchain would allow one to express designs in a widely familiar programming language, such as C or C++, and convert these into a hardware implementation that could be tested in real life, e.g. using the NetFPGA. Unfortunately, our limited testing of the existing HLS (High-Level Synthesis) tools indicated that they are definitely not ready to take an existing software system and transform that into hardware [15].

In this paper, we describe an experimental toolchain that is able to transform existing, software-oriented C++ algorithms—within the limited domain of Click-based packet processing—into a hardware description. We have chosen to work with the LLVM compiler toolkit [13], as there is a wide variety of tools and an active development community working on LLVM. This allows us to take advantage of the existing LLVM-based parallelising optimisations. LLVM modularity also allows us to easily write our own transforming compiler passes, and to add our own back end, a non-commercial HLS toolchain called AHIR [17].

The Click2NetFPGA Toolchain has a different approach than the existing HLS tools. While it also has some restrictions on what type of source code can be used as input, it transforms a complete software system into a hardware design – automatically. By selecting a restricted source domain, it was possible to have certain assumptions, and create a toolchain that could in practice

do what hasn't been done before – translating a software router to a hardware router. Our current approach combines typical software-oriented optimisations and some parallelising optimisations together with compile-time generated constant data structures and constant propagation. Our compiler front end lowers the original Click programs into versions that are more suitable for generating VHDL with the AHIR backend.

The version of our toolchain presented earlier [15] was about exploring the possibilities of some commercially available HLS tools as the backend of our toolchain. In this paper, we describe the second version of the toolchain. We have switched from a commercial backend to AHIR and rewritten the front end from scratch, building on the knowledge gained when working with the previous version. Since the second report of our work [16], we have modified the interface between Click and NetFPGA code, improved the performance and usability of the toolchain and have been able to evaluate some Click configurations on a NetFPGA card and in the Modelsim simulator.

In Section 2 we give a brief look onto the essential background information, in Section 3 we outline the operation of the toolchain, in Section 4 we go into the details with a usage example, Section 5 is on evaluation and, finally, in Section 6 we conclude the paper.

2 Background

In this section, we describe some existing related work and then we go into the major components that we have used to build our toolchain and its input and target systems.

2.1 Related Work

Over the years, several academic and commercial high-level synthesis research results and tools have been introduced. As already mentioned, most put some restrictions on the input programming language, or are solving a specific optimisation problem.

Trident is using LLVM to generate parallel hardware circuits from floating point algorithms. It doesn't allow e.g. recursion, `malloc` or `free` calls, function arguments or returned values [18].

LegUp introduces creating a soft MIPS processor and hardware accelerators that communicate through a bus interface. It uses LLVM to profile running software to identify candidate parts for hardware acceleration. The rest of the code is run as software on the MIPS processor, including code using dynamic memory, floating point and recursion [6].

The UCLA xPilot project developed a high level synthesis toolchain using LLVM [8] which evolved into a product, the AutoPilot from AutoESL, later bought by Xilinx [1]. The AutoESL High-Level Synthesis Tool accepts synthesisable ANSI C, C++ and SystemC as input.

The Click2NetFPGA Toolchain has one major difference to those listed above. It transforms a complete software system into a hardware design – automatically. Previous works either hardware accelerate certain parts of a software program, or completely synthesise only smaller units, such as single functions, that can then be used as parts of a hardware design. Click2NetFPGA takes a complete Click software router and transforms the packet processing functionality into the NetFPGA environment. By having this restricted source domain, it was possible to have certain assumptions (such as Click code having no recursion), and create the prototype toolchain.

Our toolchain is more of a "system-to-system" compiler, than yet another HLS tool. Besides the source software, the toolchain takes the system description, i.e. the Click router configration as an input. While many of the existing HLS tools could be used to compile single Click C++ elements into Verilog or VHDL, they wouldn't be able to connect these elements together nor to interface them correctly on the target system – the NetFPGA. Some of the existing HLS tools could be added to the Click2NetFPGA toolchain, either to perform a certain new optimisation, or then the toolchain could have been completely built upon some other tool than the AHIR that we chose. Before selecting AHIR, we did evaluate a number of commercial HLS tools [15]. AHIR suited our purposes better, as it is an open source project and we were free to modify it along our project.

2.2 LLVM

LLVM [13] is a collection of modular components for building compiler toolchains. The LLVM components operate on an intermediate language, called the LLVM Intermediate Representation (LLVM IR). The LLVM core consists of a compiler driver, a number of analysis and code optimisation passes, and a debugger. Several front ends and backends are using LLVM: clang [2] is intended as a replacement for GCC. The Dragonegg plugin [3] allows to replace the GCC optimisers with those from LLVM, thereby enabling all the GCC supported languages and targets to be optimised with LLVM.

LLVM has been used to implement a variety of language toolchains, including previous approaches to generate hardware [18] and bit-level optimisation of HLS data flows [19]. TCE [9] is a set of tools for designing processors based on Transport Triggered Architecture. TCE uses the LLVM clang [2] compiler as the front end for compiling hardware designs written in C and C++.

2.3 AHIR

AHIR [17] is a backend for LLVM that transforms LLVM bytecode to VHDL. It is a toolchain on its own, consisting of several different tools with intermediate result files. AHIR is still a work in progress and several of its shortcomings were discovered and fixed during this project.

The LLVM IR is first translated to an AHIR assembly language program (an **Aa** program). The **Aa** programming language corresponds to a hierarchically constructed Petri net (the type II Petri net), in which parallelism as well as complex control flow can be expressed in a direct manner. In an **Aa** program, storage variables and first-in-first-out pipes are natural objects which can be used for communication between different parts of the program. Several **Aa** programs can be linked together using an **Aa** linker.

A linked **Aa** program is then transformed to a *virtual circuit* or **vC** program, in which the control-flow, data-flow and storage aspects of the source **Aa** program are separated out (and optimised). The primary optimisations possible at this stage are: resource sharing, dependency analysis, and clustering of storage into disjoint memory spaces. The **vC** program is further translated to a VHDL description, which can be directly synthesised to target an FPGA or ASIC implementation.

The current version of AHIR supports a wide set of LLVM IR—the few notable exceptions are function pointers and recursion.

2.4 Click Modular Router

Click was introduced by Eddie Kohler [11] as a platform for developing software routers and packet processing applications. A packet processing application is assembled from a collection of simpler elements that implement basic functions, such as packet classification, queuing, or interfacing with other network devices. The elements are assembled into a directed graph using a configuration language and packets flow along the links of the graph. Click provides some features to simplify writing complex applications, including pull connections to model packet flow driven by hardware and flow-based contexts to help elements locate other relevant elements. Since its introduction, Click has been used as a tool for research into a wide variety of packet processing applications. Some representative examples are multiprocessor routers [7] and prototyping a new architecture for large enterprise networks [10].

Click modules use the full power of C++ as an object-oriented programming language, including virtual functions and dynamically allocated memory. While these language constructs facilitate code reuse and ease of programming, they complicate the task of synthesising hardware from the software.

2.5 NetFPGA

The NetFPGA platform [14] provides a platform for researchers who are interested in investigating line speed packet processing applications. It is similar to Click in the sense that it provides a set of building blocks and the user can extend the system by introducing new building blocks as well. Within this framework, students and researchers can write code to implement a variety of routing and packet processing applications that are then synthesised into hardware. The NetFPGA board can be plugged into a PC providing control plane support, and the resulting application can be tested at line speed in actual networks.

The first version of the NetFPGA platform provides a hardware board with a Xilinx Virtex-II Pro FPGA and a Verilog framework supporting hardware with a PCI bus and four 1 Gbps Ethernet ports. There is also a newer version of the NetFPGA available with four 10 Gbps Ethernet ports, a faster and bigger FPGA and more memory. We are currently using the first version of the card for this project.

3 Overview of the Toolchain

We call this prototype software a "toolchain", although it could also be called or thought of as a "compiler", as it compiles a Click router into a NetFPGA compatible hardware design. However, it is more a chain of tools: Click2LLVM, LLVM, AHIR, NetFPGA SDK, Xilinx ISE and several shell scripts, bound together by a couple of Makefiles. We are running the toolchain on Linux, but it could be ran on e.g. FreeBSD or Mac OS X with minor modifications, at least until the synthesis (Xilinx) stage, as Xilinx ISE is only available for Linux and Windows.

The Click2NetFPGA toolchain consists of five main stages:

1. Compile Click elements
2. Link required files into an LLVM Module
3. Run transformations
4. Convert to VHDL
5. Create the netlist

This process is depicted in Figure 1. The goal of the first step is to compile all Click elements and library classes into linkable object files as well as LLVM IR

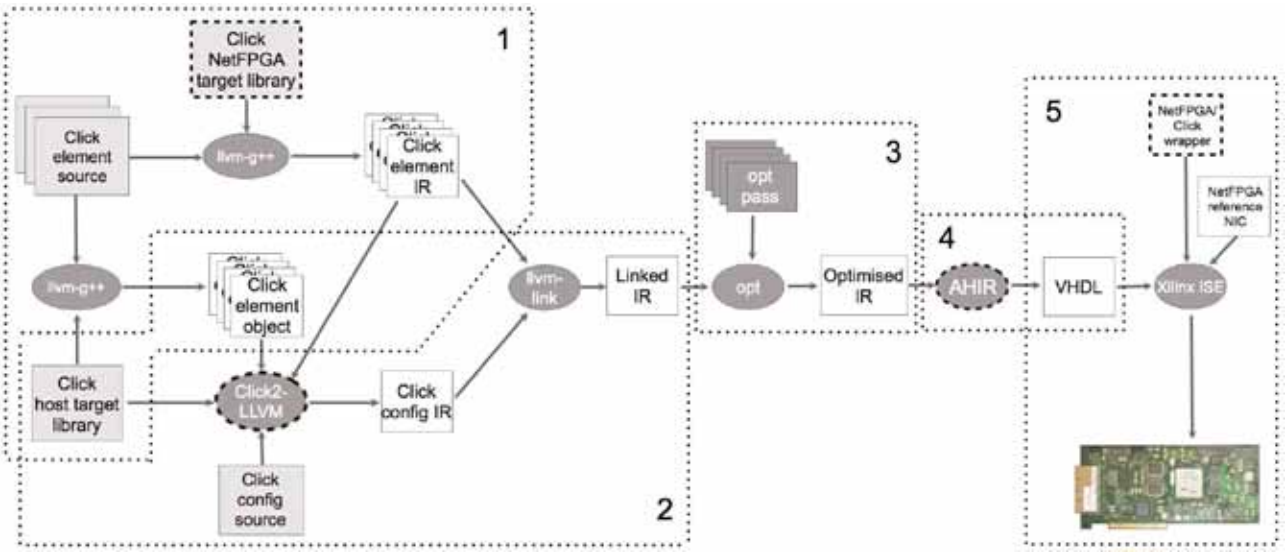


Figure 1: The Click-to-NetFPGA toolchain

source files. This stage needs to run only initially and when Click source code is modified.

In the second stage, `click2llvm` — the front end of the toolchain — reads the user-provided Click configuration file where elements and their connections are defined. The corresponding linkable object files are then loaded by the `click2llvm` tool with a Click library function.

After loading and initialising the Click router in the `click2llvm` process memory space, `click2llvm` reads the initialised values from memory and writes them out as constants in the LLVM IR format using LLVM library routines. `click2llvm` also generates some wrapper code and imports the library functions that the elements are calling. All the required code is inserted to an LLVM Module that is written out as a single LLVM IR file (Linked IR).

The resulting LLVM Module needs several transformations and optimisations so that AHIR can transform it to VHDL. First trick is to replace the *this* argument in functions with a constant variable (the Click element of the same type). For this, we have written our own LLVM transformation pass. This makes most functions constant and enables better constant propagation optimisation later. Then we run LLVM `opt` with a number of optimisation passes, including constant propagation and inlining. For inlining, we use high inline treshold to get as much inlining as possible. Finally, we run a script on the LLVM module that replaces calls to LLVM intrinsics (e.g. memcpy) with calls to our own implementations of those functions.

Finally, there will be no function calls, loops or other non-synthesisable constructs left, partly due to our several transformations, partly due to the nature of Click code. It might be possible to write e.g. recursive code in a Click element, but in practice most Click elements can be transformed into a synthesisable form. The AHIR toolchain reads this transformed LLVM IR file and creates a VHDL file.

In the last phase the toolchain takes the VHDL generated by the AHIR and combines it with a VHDL wrapper and Verilog files from the NetFPGA SDK and uses the Xilinx toolchain to create a netlist. The netlist can then be loaded to the NetFPGA. If the resulting netlist is too big to load, it will be only found out at the very end. It would be good to add a checking stage earlier in the toolchain for the required vs. available FPGA resources. This would require adding some analysis steps into the toolchain and is left for further study.

4 Implementation Details

In this section we will take a closer look on the implementation of the toolchain. First we will describe the resulting hardware architecture, and then go into more details. We'll describe the changes to Click, the step from Click to LLVM by the front end and transformations on the LLVM IR Module. Then we will explore the creation of the netlist by AHIR. At the end of the section we have a usage example on how a specific Click configuration gets compiled by the toolchain.

4.1 Resulting Hardware Architecture

The final system generated by AHIR has a single pair of input data/control pipes and a single pair of out-

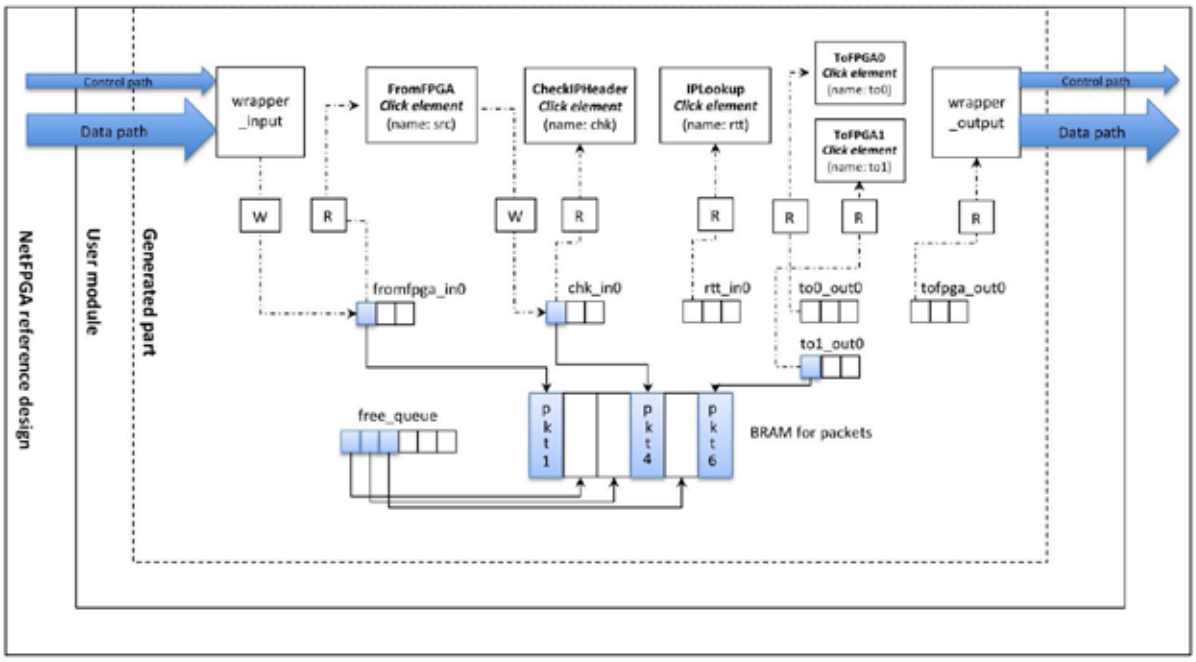


Figure 2: The resulting architecture

put data/control pipes which interface to the NetFPGA pipeline: Internally, the modules inside the AHIR system are of two types:

- Those that are "always on": these modules listen on input pipes, process data, and send processed data out on output pipes (essentially, these are pipe-line stages). Pipe accesses are blocking in nature, that is, a read from a pipe blocks until there is something in the pipe available to be read, and a write blocks until the pipe has room for new data.

- Those that are "called": these are invoked by input control signals, and normally have input and output arguments. When the module "finishes", it indicates this to the invoker, who can then use the output arguments produced by the module. These modules correspond to sub-routines invoked during system operation.

A Click configuration consists of a set of Click elements that exchange packet pointers over port connections. The pointers are used by the elements to access packets from the main memory. We have translated an input Click configuration to a set of "always on" AHIR modules where each module implements the behaviour of one Click element and the connections between Click elements are mapped to writing and reading from a named pipe that connects the elements (see Figure 2, where a "W" stands for `write_uintptr()` and an "R" for `read_uintptr()` – note that the figure represents a "snapshot of the system in time", where 3 different packets are being processed by different modules – it is not a description of the hardware with all wires drawn to be visible.). The result is a collection of interacting AHIR modules (a pipeline) which implements the original Click configuration.

The Click framework defines a packet as a distinct in-memory object that can be created, copied, and deleted. The size of the packet may change during its lifetime, which can also affect the space it occupies in the memory. In our implementation, the available memory for packets is divided into a set of fixed-size buffers. A packet inside the system is then identified by the pointer to the buffer which it occupies.

The packet buffers are managed using queues. A "free queue" (see Figure 2) contains pointers to currently free slots for packets. In addition, queues are provided between elements to hold packets that are in transit. The modules `wrapper_input()` and `wrapper_output()` are static pieces of C++ and Aa code which provide the functionality needed to obtain a free packet buffer, fill it with incoming packet data, pass the incoming packet pointer to the Click modules, receive the outgoing packet pointer from the Click pipeline, send out the outgoing packet and release the packet buffer. These are linked into the Click-based modules by the AHIR toolchain (see Figure 2). The linked design is then used to replace the output port lookup mod-

ule of the NetFPGA reference NIC design.

4.2 New Target for Click

Click consists of a runtime library and a large set of standard elements. The library implements the essential components, such as the Element, Packet, and Router classes, altogether some 80 classes. Click can be compiled as a userspace program on e.g. Linux, FreeBSD or Mac OS X, or as a kernel module for Linux. Certain parts of the library differ between these targets, e.g. in Linux kernel native `skb` structures are used to represent packets. When compiled as a userspace program, packets are stored in the memory area of the running `click` process.

We have added yet another compilation target for Click: NetFPGA. The modifications are implemented with similar precompiler instructions and often in the same functions or methods which already differed between Linux kernel and userspace implementations. One example of such a modification is the creation of a new packet through the `Packet::make()` method. In the Linux kernel an `skb` is referenced from `Class Packet`, while in userspace Click, memory is allocated dynamically from process memory space. In NetFPGA, we request a memory address for a block of pre-allocated BRAM. This BRAM is defined in C++ as an array of bytes, that is transformed to VHDL by AHIR. By default, we have allocated 32 KB of BRAM that can hold 16 packets, each having maximum size of 2 KB. A "free queue" is used to record which BRAM slots are in use and which ones are free. Access to the free queue is managed by AHIR primitives so that only one request is served at a time.

While we have modified some existing Click libraries and C++ classes, such as the Element and Packet Classes to map method implementations to AHIR primitives instead of Linux kernel or user space, our goal has been to require no modifications to the existing Click elements and to require no new guidelines for writing new Click elements. The goal is to let the programmer concentrate on describing packet processing in Click C++ without having to consider that it might be compiled to hardware instead of software—as it is of no concern whether the target platform for a software compilation would be Linux or FreeBSD.

4.3 Compiling Click Configurations

A Click configuration defines a particular assembly of Click elements, thereby constructing a packet processing application or, in Click terms, a Router. In practise, when the userlevel `click` tool is used to execute the router, the tool first parses the configuration, then initialises the router, and finally starts packet processing. In the typical case, the packet processing phase then continues until the user terminates it. In our toolchain, the first two steps of this process are performed by the `click2llvm` compiler. The last, actual packet processing step is then performed by the synthesised hardware.

When Click parses and initialises a configuration, it also instantiates all the elements defined in the configuration and invokes the initialisation methods of the resulting element instances. In practise, the elements are either statically compiled to the tool itself, or the `click` tool dynamically loads the elements into its address space from a dynamically linked shared library. The tool then instantiates the C++ classes representing the elements and invokes their virtual methods to configure and initialise them.

`click2llvm` is essentially identical with the `click` userlevel tool up to this point. While the standard tool would now initiate packet processing, our compiler writes out the resulting initialised router. For this, `click2llvm` uses LLVM libraries to link all necessary Click elements as pre-compiled LLVM modules into a single LLVM module. It then uses the LLVM `StructLayout` API to find the memory locations for different fields of the Click elements. These memory locations represent the instance variables of the C++ Click classes. It then iterates through each Click element of the Router, and writes an LLVM Global Variable for each Click element into the LLVM Module. Each element becomes a global constant struct in the resulting module.

Normally, when a packet is received in `click`, it calls the `push()` or `simple_action()` of the first element in the configuration. After finishing with the packet processing tasks, the current element then either drops the packet or calls the `push()` or `simple_action()` method of the next element in the configuration and so on. When the final element has finished the call stack returns and the first element reads the next packet. While this is a flexible way to operate in software, in hardware all Click elements will be continuously running in parallel, and we need a different approach.

In Click, connections between elements are C++ pointers. For each packet (that is not simply dropped) a Click element needs to decide the outgoing port. During the initialisation of the router, the port table of each element is populated with pointers to appropriate next elements based on the Click configuration file. In our solution, we replace the pointers with names of AHIR pipes. Instead of calling the function pointer from the port, we send the memory address of the packet through a pipe linked to the aforementioned port. We have implemented this by creating a NetFPGA specific implementation of the `Element::Port::push()`. Instead of calling the methods of the next element through pointers, we call function `write_uintptr()` which AHIR will

later interpret as a write to a named pipe. The name of the pipe is stored in the `Element::Port` Class. This way we get rid of the C++ call stack and are able to create separate hardware modules for each Click element that will run in parallel, connected by named pipes. We didn't have to implement a NetFPGA counterpart for the C++ call stack, as AHIR takes care of passing the arguments and return values for function calls.

Finally, we write out a single LLVM Module in the LLVM IR language that contains the source code for the Click elements, required parts of the Click library (such as the `Element` and `Packet` classes), constants that represent initialised elements, and a wrapper function for each element. The wrapper function maps the element to a separate program that reads packets from input ports, processes those packets and writes them to appropriate output ports (see Program 3).

Some Click elements use C++ library functions like `memcpy()`, `ntohs()` or `clock_gettime()`. The C++ compiler leaves these in the IR as "llvm intrinsics", which means that the compiler backend needs to map these calls to the target-specific implementations (on a Unix based system it would be the C++ runtime library). To achieve this, we insert LLVM instructions that implement the same functionality. For some, we currently do not have a counterpart, like for `clock_gettime()`. So far, we have implemented the required functions on-demand. It should be noted, however, that to support any (future) Click element, all system calls should be implemented on the NetFPGA. This is left as future work.

In the current prototype, we transfer all code into the NetFPGA. In practice, it would make more sense to analyse the software, and leave parts of the code to be run as software on the host CPU. This would also require automatic generation of interface code between software and hardware. However, with this prototype we have concentrated on the problem of software to hardware translation, and we leave this analysis and divisioning problem for further study.

4.4 Optimisation Phase

The LLVM Module created by the `click2llvm` tool is still unoptimised and contains constructs such as function pointers that don't easily map to hardware. We use a number of LLVM passes to transform these constructs in a more suitable form.

The first pass replaces the `this` argument in the C++ originated methods with the global variable representing the Click element. As a result the method becomes a constant function and can be inlined later. Although this means that we can have only one instance of each Click element class, the limitation can be removed, either with the trivial approach of replicating the methods and naming them differently, or writing a pass that splits the `this` argument to two: one pointing to the constant part of the class and another to the part holding instance-specific variables. The latter approach requires splitting the types in two as well. We feel this is all doable and could benefit optimising C++ software in general.

Next we pass the LLVM optimiser a list of the wrapper functions that need to be considered as the "API", and thus preserved in the output. "API" here means the same as the `main()` function in regular C/C++ programs – a starting point for program execution that is not called from within the program – that shouldn't be optimised away as "dead code". Temporary helper function definitions (`element_push()`, see Program 3) and other dead code gets optimised away and is not preserved in the output. Since the Click elements are constants, we get good results with the constant propagation and inlining passes. We use `-inline-threshold=10000` to get all packet processing code, e.g. the `push()` or `simple_action()` function of the corresponding Click element inlined in the wrapper function. We also use several other standard passes provided by the LLVM project.

4.5 Creating a Netlist

The collection of (optimised) LLVM modules produced by `click2llvm` is run through the AHIR toolchain which produces an AHIR system (described in VHDL).

Each LLVM module is implemented as an AHIR module (described as a VHDL entity/architecture pair). The AHIR module itself implements the control and data flow in the LLVM module with some optimisations; dependency analysis is used to extract parallelism from sequential statement blocks, and expensive operators (such as multipliers) are shared by multiple operations. Storage variables described in the LLVM Module collection are implemented as declared. In the AHIR system, storage variables are organised into disjoint memory spaces based on a static alias-analysis of the source program In addition, the AHIR system implements the concept of pipes (finite depth first-in-first-out queues) for inter-module communication and synchronisation. Thus, two modules in an AHIR system can communicate either through storage variables or through pipes.

The translation process in the AHIR tool-chain itself consists of three steps. In the first step, the LLVM IR is translated to an AHIR assembly level (**Aa**) program. **Aa** is an imperative and block-structured language which supports a large variety of types. The flow of control in an **Aa** program block can be specified to be sequential or parallel. During this translation:

- Each LLVM module is translated to an equivalent

Aa module. All blocks in the resulting **Aa** program are sequential in nature.

- Declared storage variables in the LLVM IR are mapped to declared storage variables in the **Aa** program.
- Pipes are inferred from the LLVM IR by keying off the special functions `uintptr read_uintptr(const char* pname)` (this is translated as a read from a pipe with the specified name) and `void write_uintptr(const char* pname, uintptr ptr)` (translated as writing the value ptr into the pipe with the specified name).

The second step is the conversion of the **Aa** program to a virtual circuit in which the control flow, data flow and storage aspects of the **Aa** program are separated. At this stage, dependency analysis is used to extract the maximum amount of parallelism that is possible from sequential statement basic-blocks. Storage variables are segregated into disjoint memory spaces whenever possible (disjoint spaces reduce load/store dependencies, and further, are accessible in parallel). The virtual circuit itself consists of distinct modules which communicate with each other through pipes or through shared memory spaces.

The final step is to generate the VHDL netlist from the virtual circuit. This translation uses a VHDL library of predesigned components such as operators, pipes, memory spaces, arbiters etc. Virtual circuit modules are translated to VHDL entities (currently, each such entity is instantiated once in the final system). Pipes are modeled in a direct manner, as are the memory spaces. Concurrency analysis is carried out in the modules to determine operations which can be mapped to the same operator without the need of arbitration. Further, depending on command line switches, the generated VHDL can be optimised for clock-period, and/or for area, or for cycle-count etc. We optimise the netlist to obtain the minimum area (given the constraints of the NetFPGA card) with a primary objective and the minimum clock period a secondary objective (in order to meet the 8 nanosecond clock period requirement of the FPGA on the NetFPGA card).

4.6 Usage Example

To illustrate the operation of the toolchain, we will go through the steps leading from the Click configuration to the optimised LLVM IR. We have created a configuration (see Program 1) which does packet switching based on the destination IPv4 address. The configuration contains seven different Click elements. FromFPGA and ToFPGA* elements in the configuration come from our own `minimal-package`, thus the `require` declaration on the first line. We currently require the elements to be introduced and given names, which is done on the next seven lines. The last five lines describe the flow of packets between the elements.

Program 1 router.click

```
require(package "minimal-package");
src :: FromFPGA;
to0 :: ToFPGA0;
to1 :: ToFPGA1;
to2 :: ToFPGA2;
to3 :: ToFPGA3;
chk :: CheckIPHeader(14);
rtt :: LinearIPLookup(172.16.0.0/24 0,
                      172.16.1.0/24 1,
                      172.16.2.0/24 2,
                      172.16.3.0/24 3);
src -> chk -> rtt;
rtt[0] -> to0;
rtt[1] -> to1;
rtt[2] -> to2;
rtt[3] -> to3;
```

Program 2 CheckIPHeader::simple_action()

```
Packet *
CheckIPHeader::simple_action(Packet *p)
{
  const click_ip *ip =
    reinterpret_cast<const click_ip *>
    (p->data() + _offset);
  unsigned plen = p->length() -
                  _offset;
  unsigned hlen, len;

  if ((int)plen <
      (int)sizeof(click_ip))
    return drop(MINISCULE_PACKET, p);
  ...
  return(p);
}
```

Elements `FromFPGA` and `ToFPGA*` are special elements that interface the Click/NetFPGA wrapper. They are for convenience (to have static wrapper code) and also as placeholders for code to transform packets between the NetFPGA and Click worlds. FromFPGA calculates the Click-specific packet lengths and offsets and stores them in the packet—this way the wrapper needs no modifications if Click itself is updated. ToFPGA does the reverse, mapping Click-specific fields into NetFPGA

control flags. We have created a separate ToFPGA element for each physical NetFPGA port: ToFPGA0 sends the packets to port 0, ToFPGA1 to port 1, and so on.

To illustrate the mapping from Click C++ code and the Click configuration file to LLVM IR, we take a closer look at one of the used elements and parts of its packet processing code. Program 2 shows a part of the C++ source code for the packet processing code of Click element `CheckIPHeader`. Method `simple_action()` is part of the Click API, and is called for the packet if the element has defined it. In the `CheckIPHeader` element, various standard checks are performed on an IP packet. A valid packet is forwarded to the next element, while packets failing a test will be dropped. The `simple_action()` function of CheckIPHeader consists of several checks, but we focus here on the first test, where the size of the IP packet is checked.

After running the `click2llvm` tool with the `router.click` configuration, we generate a wrapper function named `ahir_glue_chk()` in the resulting LLVM IR Module (see Program 3). First there is a call to `read_uintptr()` with the first argument being a pointer to the constant `@1`. This maps to a blocking read of an AHIR pipe and returns when there is something written in queue `chk_in0`. Writing to this queue is done by the `FromFPGA` element, as described in the `router.click`. Next, the read pointer is cast to type `struct.Packet`, which represents the C++ `Class Packet` of Click. Then a temporary helper function `element_push()` is called with two arguments: a pointer to the Click element (`@chk`) and the current packet (`%1`). This `ahir_glue_chk()` function becomes an "always on" AHIR hardware module – its software counterpart would be a loop that never terminates.

After optimisations on the LLVM Module, the optimised version of `ahir_glue_chk()` (see Program 4) is longer, but it has everything inlined[1]. The only calls to external functions are `read_uintptr()` and `write_uintptr()`, which are only keywords for AHIR – they will not result as function calls in hardware.

Because we have constant arguments to `element_push()`, constant propagation and inlining passes have succesfully removed it, leaving the contents of the original `simple_action()` inlined in the wrapper function. The core operation, checking that the IP packet is at least 20 bytes long, is visible in the LLVM representation before the first branch instruction (`br`). In case the packet is too short, `ahir_packet_free()` is called to free the memory slot storing the packet and `ahir_glue_chk()` returns. Otherwise, at the end of the function, `write_uintptr()` is called, leading to the `LinearIPLookup` element, as per the `router.click` configuration (see Program 1).

[1]The whole function is not presented in the listing due to space constraints.

5 Evaluation

We have compared the performance of the Click-based design vs. the Stanford reference switch implementation that is distributed with the NetFPGA software package. The former is generated with our toolchain while the latter is handwritten in Verilog. While we can see that we cannot yet reach the same performance with our toolchain, the results prove that our toolchain runs. We would like to remind that this is a proof-of-concepts prototype, yet we feel that it could be possible to reach better performance levels by further optimising our toolchain, e.g. by rewriting some Click library implementations to better match the packet processing model of the NetFPGA.

Tests performed were PPS (Packets Per Second for 98 and 1442 byte packets), Ping (average round-trip time for an ICMP Echo request/reply message pair) and maximum bandwidth (TCP for 60 seconds).

For bandwidth tests we have used iperf v2.0.4 [4] and for the ping test the standard Ubuntu Linux `ping` command. For packets per second test we have used tcpreplay v3.4.3 [5] on the sending host and iptables on the receiving host to find the approximate maximum number of packets per second before NetFPGA starts to drop packets. As the results show, we did not need more accurate measurement tools to find differences at this stage.

The test setup consisted of two standard PCs with gigabit ethernet interfaces running Ubuntu. The PCs were connected to two ports of the NetFPGA card in the third Linux machine running CentOS. With our PCs, we were able to send maximum of about 415,000 packets per second when the packet size was 98 bytes (equals 325 Mbps), and 84,000 packets when size was 1442 bytes (equals 969 Mbps).

We compared the Stanford reference switch to two different Click configurations: "router" and "pipe". Router is the configuration presented in Section 4, performing IP header checking and destination port selection based on the destination IP address. Pipe is the simplest configuration possible, it only connects NetFPGA ports in two pairs, i.e. when sending a packet to port 0, it is forwarded to port 1 and vice versa. The same handling is present for ports 2 and 3. There is no actual Click packet processing, the only elements used are FromFPGA and ToFPGA.

Analysing the results, we can see that the reference switch handles the maximum load we can feed. From previous experiments, we know that it has been designed to be running at line rate. Even though we have continuously improved our toolchain and the wrapper libraries,

Program 3 Generated ahir_glue_chk()

```
@1 = internal constant [8 x i8] c"chk_in0\00"

define void @ahir_glue_chk() {
entry:
  %0 = call i32 @read_uintptr(i8* getelementptr inbounds ([8 x i8]* @1,
                                                          i32 0, i32 0))
  %1 = inttoptr i32 %0 to %struct.Packet*
  call void @element_push(%struct.Element* getelementptr inbounds
    (%struct.CheckIPHeader* @chk, i32 0, i32 0), i32 0, %struct.Packet* %1)
  ret void
}
```

Program 4 Optimised ahir_glue_chk()

```
@1 = internal constant [8 x i8] c"chk_in0\00"
@6 = internal constant [8 x i8] c"rtt_in0\00"

define void @ahir_glue_chk() {
entry:
  %tmp13 = tail call i32 @read_uintptr(
                i8* getelementptr inbounds ([8 x i8]* @1, i32 0, i32 0))
  %tmp14 = inttoptr i32 %tmp13 to %struct.Packet*
  %tmp15 = getelementptr inbounds %struct.Packet* %tmp14, i32 0, i32 3
  %tmp16 = load i8** %tmp15, align 4
  %tmp17 = getelementptr i8* %tmp16, i32 14
  %tmp18 = getelementptr inbounds %struct.Packet* %tmp14, i32 0, i32 4
  %tmp19 = load i8** %tmp18, align 4
  %tmp20 = ptrtoint i8* %tmp19 to i32
  %tmp21 = ptrtoint i8* %tmp16 to i32
  %tmp22 = sub nsw i32 %tmp20, %tmp21
  %tmp23 = add i32 %tmp22, -14
  %tmp24 = icmp slt i32 %tmp23, 20
  br i1 %tmp24, label %"3.i1", label %"4.i"

"3.i1":
  tail call void @ahir_packet_free(i32 %tmp13)
  br label %_ZN7Element4pushEiP6Packet.exit

"4.i":
  ...
  %tmp70 = ptrtoint %struct.Packet* %tmp14 to i32
  tail call void @write_uintptr(i8* getelementptr inbounds ([8 x i8]* @6,
                                                          i32 0, i32 0),
                                  i32 %tmp70)
  br label %_ZN7Element4pushEiP6Packet.exit

_ZN7Element4pushEiP6Packet.exit:
  ret void
}
```

Measurement	Reference	Click	Percent
PPS 98 B	415K	178K	42.9
PPS 1442 B	84K	24K	28.6
Ping	105 us	115 us	109.5
Bandwidth	940 Mbps	215 Mbps	22.9

Figure 3: Reference switch vs. router.click

Measurement	Reference	Click	Percent
PPS 98 B	415K	225K	54.2
PPS 1442 B	84K	25.5K	30.4
Ping	105 us	114 us	108.6
Bandwidth	940 Mbps	227 Mbps	24.2

Figure 4: Reference switch vs. pipe.click

we lack somewhat behind. We can currently reach almost 1/3 of the 1 Gbps line rate with large packets. With smaller packets the performance is better – almost half of the line rate, as the wrapper_input needs to spend less time copying the packet to the memory subsystem. With larger packets our wrapper libraries are the bottlenecks and differences between the two Click configurations are not that large. With smaller packets, the router configuration is roughly 20 percent slower than the pipe configuration.

Based on an analysis of the pipeline stage latencies (using the Modelsim simulator and the NetFPGA verification scripts), we observe the following bottlenecks:

- The bottleneck which limits packet throughput for large packets is the interface between the NetFPGA datapath and the Click counterpart. Incoming data from the NetFPGA datapath is written into a shared packet memory, which is byte wide, single-ported and supports one access per clock cycle. Since the clock period in the NetFPGA board is set to 8 nanoseconds, this translates to an effective memory bandwidth of 1 Gbps, which is shared between reads and writes. Thus, the peak available throughput of the Click datapath is currently 500 Mbps, of which approximately 50% is actually being achieved.

- For small packets, the bottleneck (in the "router" configuration) is the latency of the IP header checking stage which was observed to be $4.2\mu s$. For short (98B) packets, this would limit the packet data rate to the 200 Mbps range, as observed. For longer packets, this bottleneck is not as serious, and the limit on the packet data rate would be higher. The latency of this stage needs to be reduced in order to achieve higher data rates (given the constraints of the NetFPGA card).

We did some experiments (using the NetFPGA verification environment together with the Modelsim simulator) to see the extent to which performance could be improved by targeting these bottlenecks. A universal method to get more performance is to use more resources, in our case FPGA slices. We could do this by replicating Click elements to parallelise some stages of the pipeline. It can be done without affecting the packet processing logic when the replicated Click element doesn't store state. As an example, we replicated the `CheckIPHeader` element to solve the first bottleneck. The result was that throughput improved by 18% over the baseline, with a corresponding increase in FPGA resource usage of 10%. It should be noted that if we create parallel paths for packets in an IP router, we need to add some packet reordering mechanism to our input and output modules, to ensure that the router operates similarly to the software version of the same Click router.

To target the packet memory bottleneck, we doubled the memory bandwidth by making it a two-banked system. With this modification together with the element replication, the throughput improvement over the baseline was 31%, with a 19% increase in FPGA resource usage. Thus, some simple bottleneck alleviation steps seem to pay good performance dividends. We have not yet tried this in actual hardware, as the FPGA chip on our NetFPGA card has limited resources, but resource replication could be useful in many applications which use larger FPGAs or ASICs.

The performance shortfall relative to hand-coded RTL can be tackled at two levels. The essential problem is to reduce the latency of the critical path through a code section. At the source-level, code transformations which enhance parallelism need to be further explored. The standard code transformations (e.g. loop unrolling, inlining) do help, but it is worth investigating if more is possible. As a fallback option, hand-optimised routines for critical code sections can make a substantial difference (analogous to the use of assembly language routines in performance critical embedded applications). To meet our goal of putting no extra burden to the software programmer, the insertion of these optimised routines should be done by the toolchain.

Further, the functionality of critical code sections can often be sub-divided into a sequence of simpler functions (pipelining the critical code segments) in order to improve the throughput of the final system. In our flow, critical code sections can be redone at three levels: at the C/C++ level, at the **Aa** level or at the VHDL level, with a corresponding performance/productivity trade-off as we descend from C/C++ to VHDL. At the hardware level, the AHIR toolchain in its current form is conservative in terms of adding register stages to meet clock period requirements (potentially adding needless cycles to the

latency). A more optimised mechanism for adding such register stages is likely to improve the latency.

Currently we initialise the Click router before hardware synthesis where we discard all configuration and initialisation related code. Therefore the resulting Click-on-NetFPGA router cannot be live reconfigured. The ideal toolchain would analyse the Click configuration and then separate parts that should be run as hardware and those that could be left as software, and then automatically create an interface between those two. That would allow parts of the Click router to run on a CPU and other parts on the FPGA, similar to the approach in [6].

6 Conclusion

In this paper we have shown that it is possible to implement a domain specific toolchain that converts high level language software to hardware. We have implemented a prototype toolchain that can transform Click routers written in C++ to a hardware description in VHDL, which can then be synthesised and run on a NetFPGA card as part of the Stanford reference NIC design.

We use the "initialise-freeze-dump" method to transform initialised C++ objects in memory to constants in the output file of the front end. This way we can run the initialisation code outside the hardware, and then use the essential parts of code directly related to packet processing to form the hardware parts. Using numerous LLVM optimisations we can then transform the code in a form that is suitable for AHIR to transform it further to VHDL.

The performance achieved by the hardware produced by this toolchain is a significant fraction of that achieved by a handwritten NetFPGA implementation. Although work remains regarding the performance and flexibility of this specific toolchain, the results in general are promising. The main performance bottlenecks are in the translation between the Click and NetFPGA packet data models. Changing the target, e.g. creating a packet processing ASIC from the Click code, and using more chip surface, would most likely bring far better performance.

7 Acknowledgments

The authors wish to thank Dr. Pekka Nikander and Dr. Sameer D. Sahasrabuddhe for their invaluable efforts and ideas in the early stages of this project.

This work has been partly funded by Tekes through its Future Internet research program.

References

[1] AutoESL High-Level Synthesis Tool. `http://www.xilinx.com/tools/autoesl.htm`.

[2] clang: a C language family frontend for LLVM. `http://clang.llvm.org/`.

[3] DragonEgg. `http://dragonegg.llvm.org/`.

[4] Iperf. `http://sourceforge.net/projects/iperf/`.

[5] Tcpreplay. `http://tcpreplay.synfin.net/`.

[6] CANIS, A., CHOI, J., ALDHAM, M., ZHANG, V., KAMMOONA, A., ANDERSON, J. H., BROWN, S., AND CZAJKOWSKI, T. LegUp: High-Level Synthesis for FPGA-based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays* (New York, NY, USA, 2011), FPGA '11, ACM, pp. 33–36.

[7] CHEN, B., AND MORRIS, R. Flexible Control of Parallelism in a Multiprocessor PC Router. In *USENIX Annual Technical Conference, General Track* (2001), Y. Park, Ed., USENIX, pp. 333–346.

[8] CHEN, D., CONG, J., FAN, Y., HAN, G., JIANG, W., AND ZHANG, Z. xPilot: A Platform-Based Behavioral Synthesis System. In *Proc. of SRC TechCon'05* (2005).

[9] JÄÄSKELÄINEN, P., GUZMA, V., CILIO, A., AND TAKALA, J. Codesign Toolset for Application-Specific Instruction-Set Processors. In *Proc. of Multimedia on Mobile Devices 2007* (2007).

[10] KIM, C., CAESAR, M., AND REXFORD, J. Floodless in Seattle: a scalable ethernet architecture for large enterprises. In *SIGCOMM* (2008), V. Bahl, D. Wetherall, S. Savage, and I. Stoica, Eds., ACM, pp. 3–14.

[11] KOHLER, E. *The Click modular router*. PhD thesis, MIT, 2000.

[12] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click modular router. *ACM Trans. Comput. Syst. 18*, 3 (2000), 263–297.

[13] LATTNER, C., AND ADVE, V. S. The LLVM Compiler Framework and Infrastructure Tutorial. In *LCPC* (2004), R. Eigenmann, Z. Li, and S. P. Midkiff, Eds., vol. 3602 of *Lecture Notes in Computer Science*, Springer, pp. 15–16.

[14] LOCKWOOD, J., MCKEOWN, N., WATSON, G., GIBB, G., HARTKE, P., NAOUS, J., RAGHURAMAN, R., AND LUO, J. NetFPGA - An Open Platform for Gigabit-rate Network Switching and Routing. In *IEEE International Conference on Microelectronics Education* (June 2007).

[15] NIKANDER, P., NYMAN, B., RINTA-AHO, T., SAHASRABUDDHE, S. D., AND KEMPF, J. Towards Software-defined Silicon: Experiences in Compiling Click to NetFPGA. 1st European NetFPGA Developers Workshop, Cambridge, UK, 2010.

[16] RINTA-AHO, T., GHANI, A., SAHASRABUDDHE, S. D., AND NIKANDER, P. Towards Software-defined Silicon: Applying LLVM to Simplifying Software. WISH - 3rd Workshop on Infrastructures for Software/Hardware co-design, Chamonix, France, 2011.

[17] SAHASRABUDDHE, S. D., SUBRAMANIAN, S., GHOSH, K. P., ARYA, K., AND DESAI, M. P. A c-to-rtl flow as an energy efficient alternative to embedded processors in digital systems. In *Proceedings of the 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools* (Washington, DC, USA, 2010), DSD '10, IEEE Computer Society, pp. 147–154.

[18] TRIPP, J. L., GOKHALE, M., AND PETERSON, K. D. Trident: From High-Level Language to Hardware Circuitry. *IEEE Computer 40*, 3 (2007), 28–37.

[19] ZHANG, J., ZHANG, Z., ZHOU, S., TAN, M., LIU, X., CHENG, X., AND CONG, J. Bit-level optimization for high-level synthesis and FPGA-based acceleration. In *FPGA* (2010), P. Y. K. Cheung and J. Wawrzynek, Eds., ACM, pp. 59–68.

Building a power-proportional software router

Luca Niccolini[‡], *Gianluca Iannaccone*[†], *Sylvia Ratnasamy*[⋆], *Jaideep Chandrashekar*[§], *Luigi Rizzo*[‡*]
[‡] *University of Pisa,* [†] *RedBow Labs,* [§] *Technicolor Labs,* [⋆] *University of California, Berkeley*

Abstract

We aim at improving the power efficiency of network routers without compromising their performance. Using server-based software routers as our prototyping vehicle, we investigate the design of a router that consumes power in proportion to the rate of incoming traffic. We start with an empirical study of power consumption in current software routers, decomposing the total power consumption into its component causes. Informed by this analysis, we develop software mechanisms that exploit the underlying hardware's power management features for more energy-efficient packet processing. We incorporate these mechanisms into Click and demonstrate a router that matches the peak performance of the original (unmodified) router while consuming up to half the power at low loads, with negligible impact on the packet forwarding latency.

1 Introduction

The network infrastructure is often viewed as an attractive target for energy-efficient design since routers are provisioned for peak loads but operate at low average utilization levels – *e.g.*, studies report network utilization levels of <5% in enterprises [28], 30-40% in large ISPs [15], and a 5x variability in ADSL networks [25]. At the same time, network equipment is notoriously inefficient at low load – *e.g.*, a survey of network devices [20, 24] reveals that the power consumed when a router is not forwarding *any* packets is between 80-90% of its peak power consumed when processing packets at full line rate. Thus although in theory networks offer significant opportunity for energy efficiencies, these savings are rarely realized in practice.

This inefficiency is increasingly problematic as traffic volumes continue their rapid growth and has led to growing attention in both research and industry [2, 9, 15, 16, 20, 24, 25, 28]. To date however, there have been no published reports on attempts to actually build an energy-efficient router. Motivated by this deficiency, and by recent advances in software routers [8, 12, 14, 17, 23], we thus tackle the question of how one might build an energy-efficient router based on general-purpose server hardware.

[*]This work was performed when the authors were affiliated with Intel Labs Berkeley.

The traditional challenge in building an energy efficient system lies in the inherent tradeoff between energy consumption and various performance metrics—in our case, forwarding rate and latency. We cannot compromise on peak forwarding rates since the incoming traffic rate is dictated by factors external to a given router (and since we do not want to modify routing protocols). We can however explore options that tradeoff latency for improved efficiency. The ability to do so requires: (1) that the underlying hardware expose primitives for low-power operation and (2) higher-level algorithms that invoke these primitives to best effect. Our work focuses on developing these higher-layer algorithms.

General-purpose hardware typically offers system designers three 'knobs' for low-power operation: (i) regulate the frequency at which individual CPU cores process work, (ii) put an idle core into a 'sleep' state (iii) consolidate packet processing onto fewer cores (adjusting the number of active cores).

As we shall see, not only do each of the above offer very different performance-vs-power tradeoffs, they also lend themselves to very different strategies in terms of the power management algorithms we must develop.

The question of how to best combine the above hardware options is, to our knowledge, still an area of active research for most application contexts and is entirely uncharted territory for networking applications. We thus start by studying the energy savings enabled by different hardware options, for different traffic workloads. Building on this understanding, we develop a unified power management algorithm that invokes different options as appropriate, dynamically adapting to load for optimal savings. We implement this algorithm in a Click software router [21] and demonstrate that its power consumption scales in proportion to the input traffic rate while introducing little latency overhead. For real-world traffic traces, our prototype records a 50% reduction in power consumption, with no additional packet drops, and only a small (less than 10 μs) cost in packet forwarding latency. To our knowledge, this is the first demonstration of an energy-efficient software router.

Before proceeding, we elaborate on our choice of general-purpose hardware as prototyping platform and how this impacts the applicability of our work. To some extent, our choice is borne of necessity: to our knowl-

edge, current network hardware does not offer low-power modes of operation (or, at least, none exposed to third-party developers); in contrast, server hardware does incorporate such support, with standard software interfaces and support in all major operating systems. Therefore, our work applies directly to software routers built on commodity x86 hardware, an area of growing interest in recent research [12, 14, 17, 23] with commercial adoption in the lower-end router market [8]. In addition network appliances – load-balancers, WAN optimizers, firewalls, IDS, *etc.* – commonly rely on x86 hardware [1, 6] and these form an increasingly important component of the network infrastructure – *e.g.*, a recent paper [31] revealed that an enterprise network of $\sim$ 900 routers deployed over 600 appliances! However, at the other end of the market, high speed core routers usually employ very different hardware options with extensive use of network processors (NPs) or specialized ASICs rather than general purpose CPUs. Hence, our results cannot be directly translated to this class of network equipment, though we expect that the methodology will be of relevance.

The remainder of the paper is organized as follows. We start with an overview of related work (§2) and a review of the power consuption of current software routers (§3) and of the power management features modern server hardware offer (§4). We continue with the study of the tradeoffs between different power management options (§5). We present the design and implementation of our unified power management algorithm in §6 and the evaluation of our prototype in §7.

2 Related Work

We discuss relevant work in the context of both networking and computer systems.

Energy efficiency in networks. Prior work on improving network energy efficiency has followed one of two broad trajectories. The first takes a network-wide view of the problem, proposing to modify routing protocols for energy savings. The general theme is to re-route packets so as to consolidate traffic onto fewer paths. If such re-routing can offload traffic from a router entirely, then that router may be powered down entirely [18, 20].

The second line of work explores a different strategy: to avoid changing routing protocols (an area fraught with concerns over stability and robustness), these proposals instead advocate changes to the *internals* of a router or switch [11,16,28,30]. The authors assume hardware support for low-power modes in routers and develop algorithms that invoke these low-power modes. They evaluate their algorithms using simulation and abstract models of router power consumption; to date, however, there has been no empirical validation of these solutions.

Our focus on building a power-proportional software router complements the above efforts. For the first line of work, a power-proportional router would enable power savings even when traffic cannot be *entirely* offloaded from a router. To the second, we offer a platform for empirical validation and our empirical results in §5 highlight the pitfalls of theoretical models.

Energy-efficient systems. With the rise of data centers and their emphasis on energy-efficiency, support for power management in servers has matured over the last decade. Today's servers offer a variety of hardware options for power-management along with the corresponding software hooks and APIs. This has led to recent *systems* work exploring ways to achieve *power proportionality* while processing a particular workload.

Many of these efforts focus on cluster-level solutions that dynamically consolidate work on a small number of servers and power-down the remaining [13, 33]. Such techniques are not applicable to our context.

Closer to our focus is recent work looking at single-server energy proportionality [19, 22, 26, 27, 34]. Some focus on improved hardware capabilities for power management [19, 22]. The remaining [26, 27, 34] focus (like us) on leveraging existing hardware support, but do so in the context of very different application workloads.

Tsirogiannis *et al.* focus on database workloads and evaluate the energy efficiency of current query optimizers [34]. The authors in [27] focus on non-interactive jobs in data centers and advocate the use of system-wide sleep during idle times, assuming idleness periods in the order of tens of milliseconds. As such, their results are not applicable to typical network equipment that, even if lightly utilized, is never completely idle. More recently, Meisner *et al.* [26] explored power management trade-offs for online data-intensive services such as web search, online ads, *etc*. This class of workloads (like ours) face stringent latency requirements and large, quick variations in load. The authors in [26] use benchmarks to derive analytical models, based on which they offer recommendations for future energy-efficient architectures.

In contrast to the above, we focus on networking workloads and exploiting low-power options in current hardware. The result of our exploration is a lightweight, online, power saving algorithm that we validate in a real system. We leave it to future work to generalize our findings to application workloads beyond networking.

3 Deconstructing Power Usage

A necessary step before attempting the design of a power-proportional router is to understand the contribution of each server component to the overall power usage.

Server architecture. For our study, we chose an off-the-shelf server based on the Intel Xeon processor that

is commonly used in datacenter and enterprise environments. Our server has two CPU processors each of which consists of six cores packaged onto a single die.

The two processors are Xeon 5680 "Westmere" with a maximum clock frequency of 3.3 GHz. They are connected to each other and to the I/O hub via dedicated point-to-point links (called QuickPath Interconnect in our case). The memory (6 chips of 1 GB) is directly connected to each of the processors. We equipped the server with two dual-port Intel 10Gbps Ethernet Network Interface Cards (NICs). The I/O hub interfaces to the NICs (via PCIe) and to additional chipsets on the motherboard that control other peripherals. Other discrete components include the power supply and the fans.

From a power perspective, we consider only the components that consume a non-negligible amount of power: CPUs, memory, NICs, fans, and the motherboard. We use the term "motherboard" as a generic term to include components like the I/O Hub and PCIe bridge. It also includes other system peripherals not directly used in our tests: USB controller, SATA controller, video card, and so forth. The power supply unit (PSU) delivers power to all components using a single 12V DC line to the motherboard, which is in turn responsible for distributing power to all other subsystems (CPUs, fans, *etc.*).

We measure the current absorbed by the system by reading the voltage across 0.05Ω shunt resistors placed in series to the 12V line. This gives us the ability to measure power with good accuracy and sampling frequency and bypassing the PSU.

Workload. Our server runs Linux 2.6.24 and Click [21] with a 10G Ethernet device driver with support for multiple receive and transmit queues. Using multiple queues, a feature available in all modern high speed NICs, we can make sure each packet is handled from reception to transmission by only one core without contention. We use Receive Side Scaling (RSS) [5] on the NIC to define the number of queues and, hence, the number of cores that will process traffic. RSS selects the receive queue of a packet using the result of a hash computed on the 5-tuple of the packet header. This way traffic is evenly spread across queues (and cores). In the rest of the paper, when we refer to a number of cores n we also imply that there are n independent receive queues.

In the following sections, we first focus on the performance and power consumption of a single server operating as an IPv4 router with four 10G ports. Later, in §7, we consider more advanced packet processing applications such as NetFlow, AES-128 encryption and redundancy elimination [10]. For traffic generation we use two additional servers that are set up to generate either synthetic workloads with fixed size packets or trace-driven workloads. The two machines are identical to our router server and each sends packets on two 10Gbps interfaces.

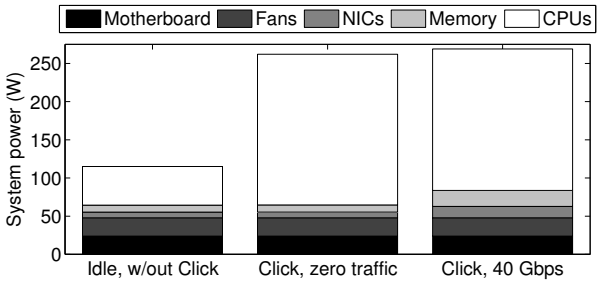


Figure 1: *Breakdown of power consumption across various components when the system is idle and Click is not running (left), with Click running and no traffic (middle), and with Click forwarding 40 Gbps of traffic (right).*

Power characterization. Isolating the power consumption of each component is useful to answer two different questions: (1) where does the power go in a modern server? (2) how does the power consumption vary with load?

The answer to the first question will let us identify which components are the largest consumers while the answer to the second tells us how much can be saved by making that component more energy-proportional.

We measure power in three sets of experiments: i) an idle system without Click running; ii) a system running Click with no input traffic; iii) a system running Click and forwarding 40 Gbps over four interfaces.

Given that we have only one aggregate power measurement for the entire server (plus one dedicated to the NICs), we need to perform several measurements to isolate each component. The system has eight fans, two CPUs and six memory chips. For each scenario we measure the power P with all components and then we physically remove one component (a memory chip, a CPU or a fan) and then measure the power P'. The difference $P - P'$ is then the power consumed by that component.

Figure 1 shows the results of our experiments with IPv4 routing. At peak the system reaches 269 W – 2.3x the idle power consumption of 115 W. With Click running and no traffic the power consumption is 262 W, which is less than 3% lower than the peak power. Several design considerations stem from the results in Figure 1:

The CPUs are clearly the dominant component when it comes to power usage. Even when all cores are idle, they consume almost half of the total idle power, or $\approx$ 25 W each. That energy is mostly used to keep the "uncore" online. At peak, the CPUs reach $\approx$ 92 W each, which is about four times their idle power; this contributes to more than doubling the total power compared to idle.

The other system components contribute little to the over-

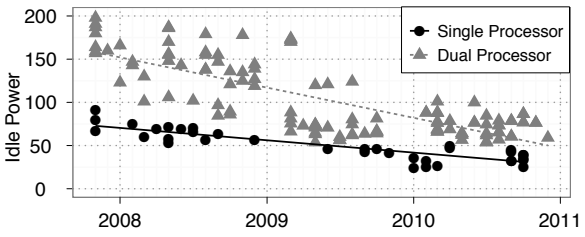


Figure 2: *Server idle power over time (data from [32])*

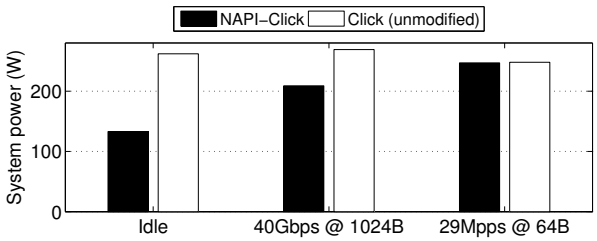


Figure 3: *Power consumption of NAPI-Click vs Click in polling mode.*

all power consumption. The motherboard absorbes a constant power (24 W). The memory and NICs exhibit a significant dynamic range but the overall contribution is quite limited (20 W for the six memory chips and 15 W for the four 10 Gbps interfaces).

The system idle power is relatively high at 115 *W*. This is an ongoing concern for system architects and the general trend is towards a reduction in idle power. Figure 2 shows the idle power of a number of systems since 2007 as reported by SpecPower [32]. The plot shows a clear downward trend in idle power, that is expected to continue thanks to the integration of more components into the CPUs and more efficient NICs [9] and memory [7].

Overall, our analysis indicates that to achieve energy efficiency we should focus on controlling the CPUs as they draw the most power and exhibit the largest dynamic range from idle to peak.

Addressing Software Inefficiencies. From Figure 1 we saw that the software stack exhibits very poor power scaling: the total power consumption is almost constant at zero and full load. A pre-requisitive to exploiting power management features of the server is to ensure that the software stack itself is efficient – *i.e.*, doesn't engage in needless work that prevents the creation of idle periods.

In this case, the poor power scaling is caused by Click that disables interrupts and resort to polling the interface to improve packet forwarding performance. An alternative solution is deploy the same mechanisms that can be found in Linux-based systems under the name of Linux NAPI framework. In that framework, interrupts still wake up the CPU and schedule the packet processing thread. That thread then disables the interrupts and switches to polling until the queue is drained or a full batch of packets is processed. It is well understood that this approach will lead to power savings as it gives the CPU an opportunity to transition to a low power state when interrupts are enabled. However, there is no study that show the forwarding latency penalty of using interrupts. To this end, we modified the 10G ethernet driver and Click to operate under the Linux NAPI framework and run power and latency measurements. We call this new codebase NAPI-Click.

Figure 3 shows the power savings with NAPI-Click *vs.* unmodified polling-mode Click for different packet rates. We see that NAPI-Click reduces power consumption by 50% in idle mode and 22% when forwarding 40 Gbps of 1024B packets. On the performance side, enabling interrupts has no impact on the maximum packet rate the router can forward – 29 Mpps with 64B packets.

Regarding latency, we measured a packet forwarding latency across the server of about 7 μs with polling and 12 μs with NAPI-Click. At high load, the latencies of the two methods tend to converge as NAPI-Click degenerates to polling at high load. In the interest of space, we refer the reader to [29] for a more exhaustive description of our setup and analysis of the latency results.

4 Server Support for Power Management

Modern servers provide a variety of power management mechanisms that operate at all levels in the platform. Here, we summarize the controls exposed to the OS.

Core idle states (C-states). Each core in a package can be independently put into one of several *low power* or "idle" states, which are numbered from C1 to Cn. The state C0 refers to a core that is executing instructions and consuming the highest amount of power. Higher numbered states indicate that a successively larger portion of the core is disabled resulting in a lower power draw.

A core enters into a C state either by executing a HALT or MONITOR/MWAIT instruction. C states are exited (to return to the C0 state) when the desired event (interrupt, or write access to the monitored memory range) occurs. With a larger amount of circuitry turned off, higher idle states incur a higher *exit latency* to revert back to the fully operational C0 state. We measure exit latencies for C states in §5.

As an example, the processors we use offer three C states: C1, C3 and C6. A core in C1 is *clock gated*, in C3 the L1 and L2 caches are flushed and turned off while in C6 power distribution into the core is disabled and the core state is saved in the L3 cache.

Processor performance states (P-states). While a processor core is in active C0 state, its power consumption is determined by the voltage and operating frequency, which can be changed with Dynamic Voltage and Frequency Scaling (DVFS). In DVFS, voltage and frequency are scaled in tandem. Modern processors offer a number of frequency and voltage combinations, each of which is termed a P state. P0 is the highest performing state while subsequent P states operate at progressively lower frequencies and voltages. Transitions between P states require a small time to stabilize the frequency and can be applied while the processor is active. P states affect the entire CPU and all cores always run at the same frequency. The processor in our system offers 14 P states with frequencies ranging from 1.6 GHz to 3.3 GHz.

5 Studying the Design Space

We now turn to exploring the design space of power-saving algorithms. A system developer has three knobs by which to control CPU power usage: i) the number of cores allocated to process traffic; ii) the frequency at which the cores run; and iii) the sleep state that cores use when there is no traffic to process.

The challenge is how and when to use each of these knobs to maximize energy savings. We first consider the single core case and then extend our analysis to multiple cores. Our exploration is based on a combination of empirical analysis and simple theoretical models; the latter serving to build intuition for why our empirical results point us in a particular direction.

5.1 Single core case

With just one core to consider our design options boil down to one question: is it more efficient to run the core at a lower frequency for a longer period of time *or* run the core at a (relatively) higher frequency for a shorter period of time and then transition to a sleep state?

To understand which option is superior, we compare two extreme options by which one might process W packets in time T:

- **the "hare"**: the core runs at its maximum frequency, f_{max} and then enters a low-power sleep state (C1 or below). This strategy is sometimes referred to as "race-to-idle" as cores try to *maximize* their idle time.
- **the "tortoise"**: the core runs at the minimum frequency, f_x, required to process the input rate of W/T. I.e., we pick a core frequency such that there is *no* idle time. This is a form of "just-in-time" strategy.

To compare the two strategies we can write the total energy consumption of one core over the period T as:

$$E = P_a(f)T_a(W, f) + P_sT_s + P_iT_i, \qquad (1)$$

where $T = T_a(W, f) + T_s + T_i$. The first term $P_a(f)T_a(W, f)$ accounts for the energy used when actively processing packets. $P_a(f)$ is the active power at frequency f and $T_a(W, f)$ is the time required to process W packets at frequency f. The second term is the energy required to transition in and out of a sleep state, while the third is the energy consumption when idle.

With the *tortoise* strategy, the core runs at a frequency $f = f_x$ such that $T = T_a(W, f_x)$ and no idle time; hence:

$$E_{tortoise} = P_a(f_x)T_a(W, f_x) \qquad (2)$$

With the *hare* strategy, $f = f_{max}$ and hence:

$$E_{hare} = P_a(f_{max})T_a(W, f_{max}) + P_sT_s + P_iT_i, \quad (3)$$

To facilitate comparison, let's assume (for now) that $T_s = 0$ (instantaneous transitions to sleep states) and $P_i = 0$ (an ideal system with zero idle power). Note that these assumptions greatly favor any *hare*-like strategy.

With these assumption the comparison between the *tortoise* and *hare* boils down to a comparison of their $P_a()T_a()$ terms in Equations 2 and 3. The literature on component-level chip power models tell us that the active power $P_a(f)$ for a component using frequency/voltage scaling grows faster than $O(f)$ but slower than $O(f^3)$.[1] The term $T_a(W, f)$ instead scales as $1/f$ in the best case. Hence, putting these together, we would expect that $P_a(f_{max})T(W, f_{max})$ is always greater than $P_a(f_x)T(W, f_x)$, since $f_{max} \geq f_x$. Hence, despite our very '*hare* friendly' assumptions ($T_s = P_i = 0$), basic chip power models would tell us that it is better to behave like a *tortoise* than a *hare*.

Do experimental results agree with the above reasoning? To answer this, we use the same experimental setup as in §3 with a workload of 64B packets and plot results for our server using between 1 to 12 cores. We show results with more than just one core to ensure we aren't inadvertently missing a trend that arises with multiple cores; in all cases however we only draw comparisons across tests that use the same number of cores.

We first look at how $P_a(f)$ scales with f in practice. Fig. 4 plots the active power consumption, $P_a()$, as a function of frequency. For each data point, we measure power at the maximum input packet rate that cores can sustain at that frequency; this ensures zero idle time and hence that we are indeed measuring only $P_a()$. We see that $P_a()$ does not grow as fast as our model predicted – *e.g.*, halving the frequency leads to a drop of only about 5% in power usage with one core, up to a maximum of 25% with twelve cores. The reason is that, in practice,

[1] This is because power usage in a chip can be modeled as cV^2f where c is a constant that reflects transistor capacitance, V is the voltage and f is the frequency. As frequency increases voltage must also increase – larger currents are needed to switch transistors faster.

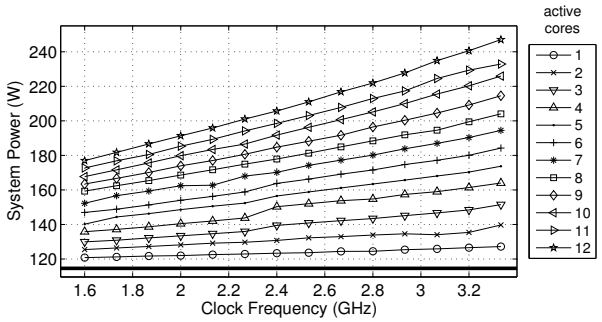


Figure 4: *Power vs. Frequency at maximum sustained rate. The solid black line marks the system idle power.*

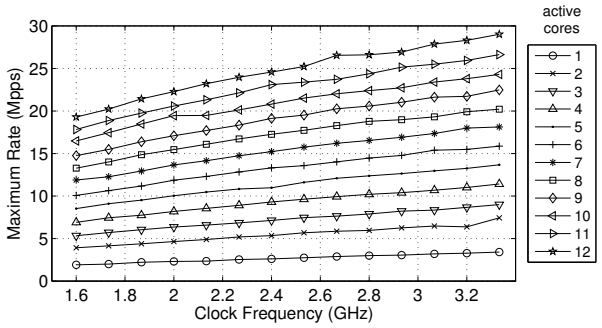


Figure 5: *Packet rate vs. frequency at maximum sustained rate for different number of cores.*

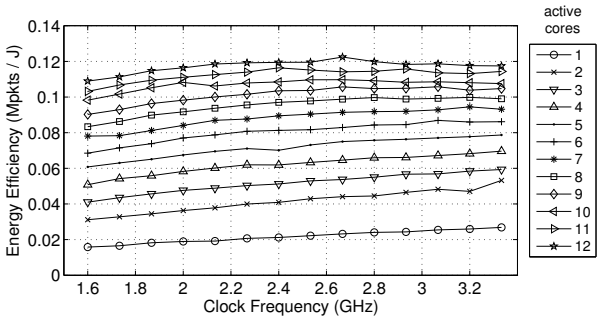


Figure 6: *Processed packets per Joule varying the frequency and the number of cores.*

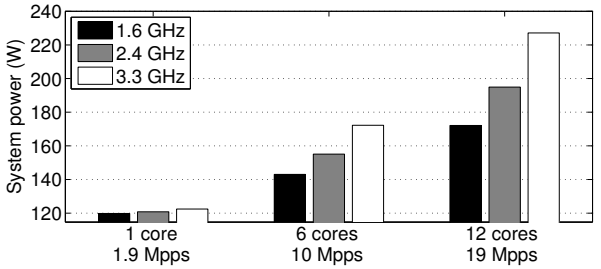


Figure 7: *Power consumption of three frequencies for a constant input rate. At the lowest frequency (black bar) the system has no idle time.*

many of the server's components, both within the CPU (*e.g.*, caches and memory controllers) and external to the CPU (*e.g.*, memory and I/O subsystems) are not subject to DVFS leading to lower savings than predicted.

Thus, active power consumption grows much more slowly with frequency than expected. What about $T_a(W, f)$? Since the processing time dictates the forwarding rate a core can sustain, we look at the forwarded packet rate corresponding to each of the data points from Fig. 4 and show the results in Fig. 5. Once again, we see that our model's predictions fall short: doubling the frequency does not double the sustainable packet rate. For example, with one core, doubling the frequency from 1.6 GHz to 3.2 GHz leads to an increase of approx. 70% in the forwarded packet rate (down to 45% with twelve cores). Why is this? Our conjecture is that the perfect $1/f$ scaling of processing time applies only to a CPU-intensive workload. Packet processing however is very memory and I/O intensive and access times for memory and I/O do not scale with frequency. Therefore, as we increase the frequency we arrive at a point where the CPU does its work faster and faster but it is then stalled waiting for memory accesses, leading to a point where increasing the frequency no longer improves productivity.

In summary, we find that, $P_a()$ grows more slowly than expected with frequency but at the same time $T_a()$ decreases more slowly than expected. Where does this leave the product $P_a()T_a()$? We cannot directly measure this product (energy) and hence we look at efficiency in terms of packets-forwarded per Joule, obtained by dividing the maximum sustained forwarding rate (from Fig. 5) by the power consumption (Fig. 4). The result is shown in Fig. 6 for increasing frequency. As before, this captures system efficiency while actively processing work (*i.e.*, there's no idle time). We see that, empirically, running at the maximum frequency is marginally more energy efficient in all configurations. That is, if our assumptions of $T_s = P_i = 0$ were to hold, then the *hare* would actually be marginally *more* power-efficient than the *tortoise* – counter to theoretical guidelines.

Of course, our assumptions are not realistic. In particular, we saw earlier (Fig. 3) that P_i is quite high. Hence, the consumption due to the P_i and P_s terms in Eqn. 3) tilts the scales back in favor of the *tortoise*.

The final validation can be seen in Fig. 7 where we plot the total power consumption for a fixed input packet rate at different frequencies. These are the first set of results (in this section) where we include idle and transition times. Based on our analysis from the following section, we make an idle core enter the C1 sleep state. We consider 1, 6 and 12 cores and, for each experiment, we fix the input packet rate to be the maximum rate the system can sustain at a frequency of 1.6 GHz; *i.e.*, at 1.6 GHz,

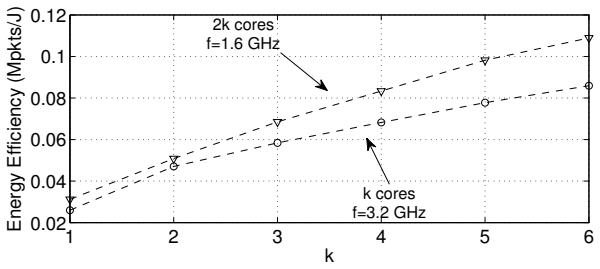


Figure 8: *Comparing processed packets per Joule for k cores at frequency f and nk cores at frequency $\frac{f}{n}$.*

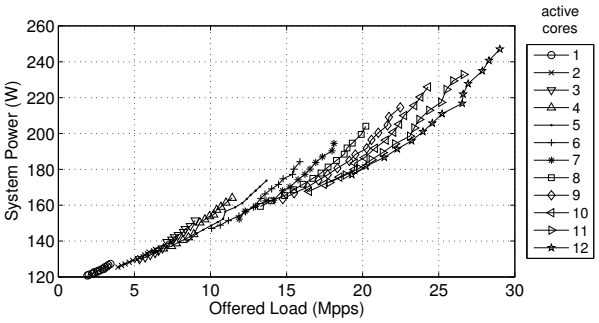


Figure 9: *System power varying number of cores and operating frequency. Each point corresponds to the maximum 64B packet rate that configuration can sustain.*

there is no idle time and this corresponds to the *tortoise* strategy; the tests at 2.4 GHz and 3.3 GHz represent the (fast and fastest) *hare* strategy. It is clear from the figure that the configuration with the lowest frequency is always the most energy efficient.

Hence, in summary, the *tortoise* wins in keeping with what the theoretical power model would suggest. However this is *not* because the work is more efficiently processed at low frequency (as the power models would indicate) but because being idle is not sufficiently efficient (*i.e.*, the terms $P_i T_i$ and $P_s T_s$ are quite significant).

5.2 Multiple cores

We now consider the case where we have N cores to process W packets in time T. The results in the previous section show that it is more energy efficient to run an individual core at a low frequency and minimize idle time. Applying this to the multiple cores case would mean dividing the work across multiple cores such that each core runs at a frequency at which it is fully utilized (*i.e.*, no idle time). In doing so, however, we must decide whether to divide the work across fewer cores running at a (relatively) higher frequency *or* more cores at a low frequency. Again, we first look to theoretical models to understand potential trade-offs.

Let us assume that a single core at a frequency f can process exactly W/k packets in time T. Then, our design choice is between:

- **"strength in speed":** use k cores at a frequency f. Each core processes W/k packets, sees no idle time, and hence consumes energy $kP_a(f)T(W/k, f)$;
- **"strength in numbers":** use nk cores at a frequency f/n. Each core now processes W/nk packets, sees no idle time, and hence consumes energy $nkP_a(f/n)T(W/kn, f/n)$.

As before, an idealized model gives $T(W/k, f) = T(W/kn, f/n)$ (since cores at f/n process $1/n$-th the number of packets at $1/n$-th the speed) and hence our comparison is between $kP_a(f)$ and $nkP_a(f/n)$. If the active power $P_a(f)$ grows faster than $O(f)$ (as the theory suggests) then the strength-in-numbers approach is clearly more energy efficient. This points us towards running with the maximum number of cores, each running at the minimum frequency required to sustain the load.

Looking for empirical validation, we consider again the packets forwarded per Joule but now comparing two sets of configurations: one with k cores running at frequency f and one with nk cores at frequency f/n. Unfortunately, since our hardware offers a frequency range between $[1.6, 3.3]$ GHz, we can only derive data points for $n = 2$ and $k = [1, 6]$ – the corresponding results are shown in Fig. 8. From the figure we see that the empirical results do indeed match the theoretical guidelines.

However, the limited range of frequencies available in practice might raise the question of whether we can expect this guideline to hold in general. That is, what can we expect from m_1 cores at frequency f_1 *vs.* m_2 cores at frequency f_2 where $m_1 < m_2$ and $f_1 > f_2$? To answer this, we run an exhaustive experiment measuring power consumption for all possible combinations of number of cores (m) and core frequency (f). The results are shown in Fig. 9 – for each $\langle m, f \rangle$ combination we measure the power consumption (shown on the Y-axis) and maximum forwarding rate (X-axis) that the m cores can sustain at a frequency f. Thus in all test scenarios, the cores in question are fully utilized (*i.e.*, with no idle times). We see that, for any given packet rate, it is always better to run more cores at a lower frequency.

Applying this strategy under the practical constraints of a real server system however raises one additional problem. A naive interpretation of the strength-in-numbers strategy would be to simply turn on all N available cores and then crank up/down the frequency based on the input rate — *i.e.*, without ever worrying about how many cores to turn on. This would be reasonable if we could tune frequencies at will, starting from close to 0 GHz. However, as mentioned, the frequency range available to us starts at 1.6 GHz and, so far, we've only

sleep state	system power	avg. exit latency
C1	133 W	$< 1\ \mu$s
C3	120 W	60 μs
C6	115 W	87 μs

Table 1: Power consumption and exit latency for different sleep states. See [29] for details on the experimental setup used to measure the above.

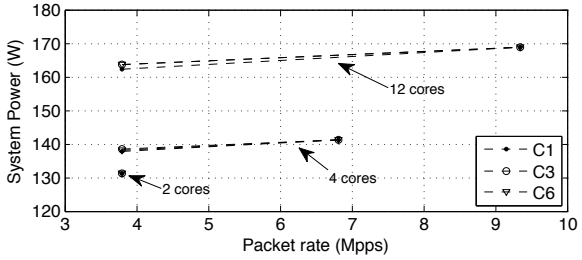


Figure 10: *Comparing the impact of different C states on power consumption. All cores are running at* 1.6 *GHz.*

considered the efficiency of m cores that run *fully utilized* at any frequency. For example, if we consider the 12 core case in Fig. 9 we only see the power consumption for data rates higher than approx. 19Mpps (the max forwarding rate at 1.6 GHz). How would 12 cores fare at lower packet rates?

Before answering this question, we must first decide how to operate cores that are under-utilized *even at their lowest operating frequency*. The only option for such cores is to use sleep states and our question comes down to which of the three sleep states – C1, C3 or C6 – is best. This depends on the power-savings *vs.* exit-latency associated with each C state. We empirically measured the idle power consumption and average transition (a.k.a 'exit') latency for each C state – the results are shown in Table 1. We see that C3 and C6 offer only modest savings, compared to C1, but incur quite large transition times. This suggests that C1 offers the 'sweet spot' in the tradeoff.

To verify this, we measured the power consumption under increasing input packet rates, using a fixed number of cores running at a fixed frequency. We do three set of experiments corresponding to whether an under-utilized core enters C1, C3 or C6. Fig. 10 shows the results for 2, 4, and 12 cores and a frequency of 1.6 GHz. We see that which C-state we use has very little impact on the total power consumption of the system. This is because, even at low packet rates, the packet-interarrival times are low enough (*e.g.*, 1 Mpps implies 1 packet every 1μs) that cores have few opportunities to transition to C3 or C6. In summary, given its low transition time, C1 is the best choice for an under-utilized core.

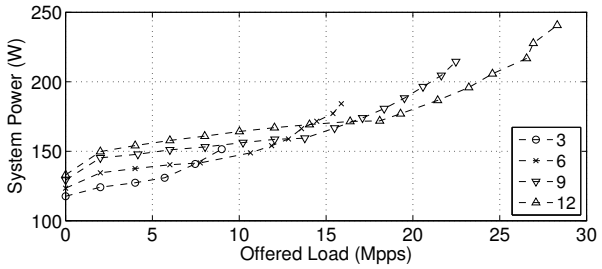


Figure 11: System power consumption varying the input data rate. Cores transition to C1 when underutilized.

Now that we know what an under-utilized core should do, we extend the results from Fig. 9 by lowering the input data rates to consider the case of under-utilized cores. For clarity, in Fig. 11 we plot only a subset of the data points. We see that, for any packet rate, the appropriate strategy is *not* to run with the maximum cores at the lowest frequency but rather to run with the maximum cores that can be kept *fully utilized*. For example, at 5 Mpps, the configuration with 3 active cores saves in excess of 30 W compared to the 12 cores configuration.

In summary, our study reveals three key guidelines that maximize power savings:

1. **strength in numbers:** the aggregate input workload should be equally split between the maximum number of cores that can be kept fully utilized.

2. **act like a tortoise:** each core should run at the lowest possible frequency required to keep up with its input workload.

3. **take quick-n-light naps:** if a single core running at its lowest possible frequency is under-utilized, it should enter the lightest C1 sleep state.

Following the above guidelines leads to the lower envelope of the curves in Fig. 9 which represents the optimal power-consumption at any given packet rate. In the following section, we describe how we combine these guidelines into a practical algorithm.

6 Implementation

Our empirical results tell us that a strategy that yields the maximum power saving is one that tracks the lower envelope of the curves in Figure 9. A simple implementation of that strategy could be to use a lookup table that, for any input data rate, returns the optimal configuration. However, such an approach has two limitations. First, perfect knowledge of instantaneous data rate is required. Second, any change in the hardware would require comprehensive benchmarking to recompute the entire table.

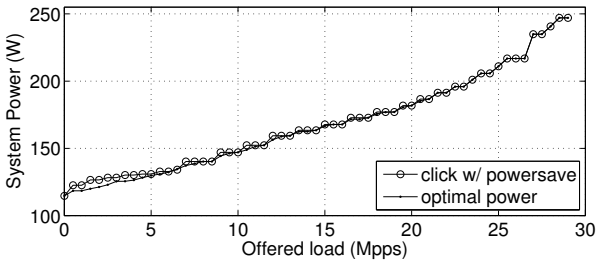


Figure 12: *Power usage as a function of packet rate for our power saving algorithm and an optimal strategy.*

An alternative is to devise an algorithm that tries to approximate the lower envelope of the curves with no explicit knowledge of those curves. An approach that maps quite well to the guidelines is the following iterative algorithm (that we will call "PowerSave"):

- Start with only one core active at the minimum frequency. All other cores are in C6 deep power down state ("inactive").
- As the load increases wake up one more core and split the traffic evenly among all active cores. If all cores are active, increase the frequency of all cores to keep up with the input data rate.
- If traffic load decreases, first lower the frequency of all cores and then start turning off one core at a time consolidating traffic onto the remaining active cores.

To understand how well this algorithm could approximate the optimal power curve we emulate its behavior by using the power measurement used to plot Figure 9. In Figure 12, we show the power usage for an optimal algorithm, that follows the lower envelope of Figure 9, and for our PowerSave algorithm. The approximation closely tracks the optimal algorithm. At very low rate our algorithm chooses to turn two cores on much earlier than the optimal solution would. Even so, the additional power consumption is very small (below 5 W) given that the two cores always run at the lowest frequency.

Turning this algorithm into an actual implementation requires two mechanisms: i) a way of determining whether the current configuration is the minimum required to sustain the input rate, and ii), a way of diverting traffic across more cores or consolidating traffic onto fewer cores. The remainder of this section describes our implementation of the two mechanisms.

6.1 Online adaptation algorithm

The goal of the adaptation algorithm is to determine if the current configuration is the minimum configuration that can sustain the input traffic rate. The algorithm needs to be able to quickly adapt to changes in the traffic load and do so with as little overhead as possible.

A good indicator of whether the traffic load is too high or low for the current configuration is the number of packets in the NIC's queues. Obtaining the queue length from the NICs incurs very little overhead as that information is kept in one of the NIC's memory-mapped registers. If the queues are empty – and stay empty for a while – we can assume that too many cores are assigned to process the incoming packets. If the queues are filling up instead, we can assume that more cores (or a higher frequency) are needed to process the incoming packets.

This load estimation is run as part of the interrupt handler to allow for a quick response to rate changes. Only one core per interface needs to run this estimator to decide when to wake up or put to sleep the other cores. We set two queue thresholds (for hysteresis) and when the number of packets is below (above) a threshold for a given number of samples we turn off (turn on) one core The pseudocode of the algorithm is the following:

```
for (;;) {
   // get a batch of packets and queue length
   qlen, batch = input(batch_len);

   // now check current queue length
   if (qlen > q_h) {
      // the queue is above the high threshold.
      c_l = 0; c_h++;
      if (c_h > c_up) {
         c_h = 0;
         <add one core, if all are on, raise frequency>
      }
   } else if (qlen < q_l) {
      // the queue is below the low threshold
      c_l++; c_h = 0;
      if (c_l > c_down) {
         c_l = 0;
         <decrease frequency, if at min remove one core>
      }
   } else { c_h = c_l = 0; }
   // process a full batch of packets
   process(batch);

   if (qlen == 0) halt();
}
```

The parameters q_h and q_l are used to set the target queue occupancy for each receive queue. Setting low queue thresholds leads to smaller queueing delays at a cost of higher power usage. The choice of the most appropriate thresholds is best left to network operators: they can trade average packet latency for power usage. In our experiments we set $q_h = 32$ and $q_l = 4$ for a target queueing delay of ≈ 10 μs – assuming a core can forward at 3 Mpps.

The variables c_h and c_l keep track of how many times the queue size is continuously above or below the two thresholds. If the queue is above the q_h threshold for a sufficient time, the system is under heavy load and we add one more core or increases the frequency. Conversely, if we are under the q_l threshold for a sufficient time, we decrease the frequency or turn off one core. Note that every time we change the operating point, the counter is reset to give the system sufficient time to react to the change. After adjusting the operating level, we

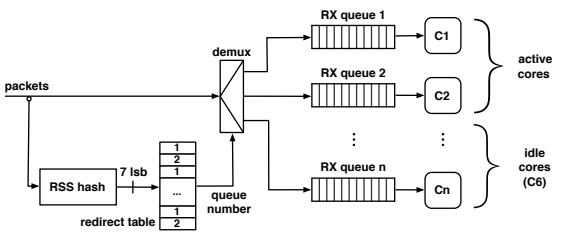


Figure 13: *NIC queues and redirect table. Hashes are computed on the 5-tuple of each packet, and used to address one of the 128 entries of the redirect table.*

process a batch of packets, and either repeat the loop, or halt (thus moving to state C1) if there are no more packets to be processed. In case of a halt, the next interrupt will resume processing with no further delays other than those related to interrupt generation and dispatching.

6.2 Assigning queues to cores dynamically

Our algorithm requires that the NIC be able to split the incoming traffic evenly between a variable set of cores. For this we use the Receive Side Scaling (RSS) [5] mechanism that implement the support for multiple input and output queues in modern NICs.

For each incoming packet, the NIC computes a hash function on the five-tuple (source and destination IP, source and destination port, and protocol number) of the packets. The 7 least significant bits of the hash are used as an index into a *redirect table*. This table holds 128 entries that contain the number of the queue to be used for packets with that hash, as shown in Figure 13. Each queue has a dedicated interrupt and is assigned to one core. The interrupt is raised on packet reception and routed to the associated core.

The number of queues on a NIC can only be set at startup. Any change in the number requires a reset of the card. For this reason we statically assign queues to cores as we start Click. To make sure inactive cores do not receive traffic, we modify the redirect table by mapping entries to queues that have been assigned to active cores.The entries are kept well balanced so that traffic is spread evenly across active cores. The table is only 128 bytes long, so modifications are relatively fast and cheap.

If a core does not receive traffic on its queue it remains in deep power down. When additional cores are needed to process traffic, we reconfigure the redirect table to include the additional queue. The update to the redirect table has almost immediate effect; this is important because it immediately reduces the load on the active cores. As soon as the first packets arrives to the new queue, the interrupt wakes up the corresponding core, which eventually (after the exit latency) starts processing the traffic.

Taking out a core is equally simple: we reprogram the redirect table so that no new packets will reach the extra core. Also, we instruct the core to enter a C6 state on `halt()` instead of C1. Eventually, the queue will become empty, and the core will enter C6 from which it does not exit until the queue is enabled again.

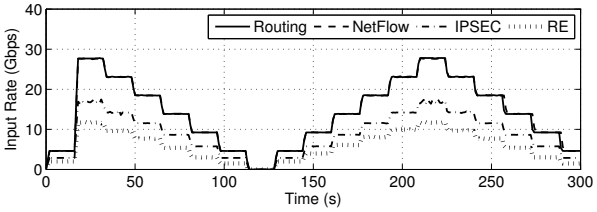


Figure 14: *Input traffic profile for different applications.*

7 Evaluation

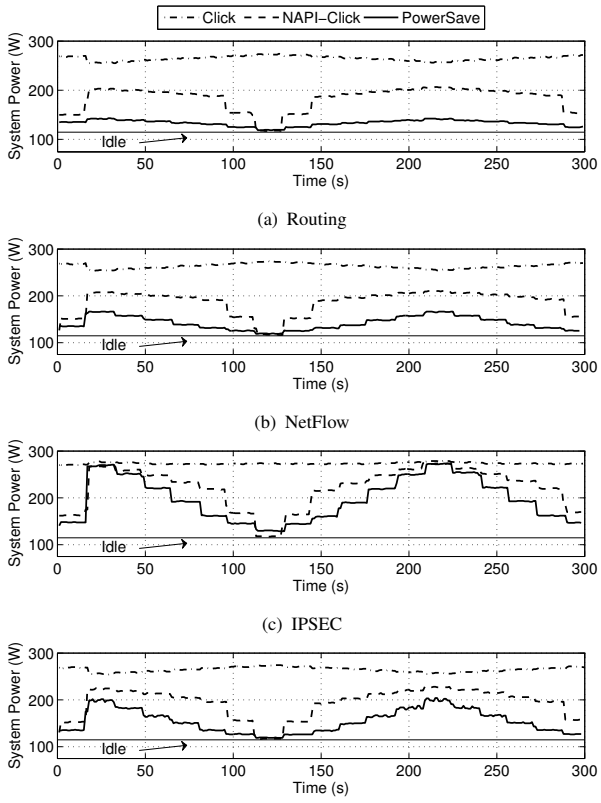
To evaluate the performance of our prototype router in a realistic setting we generate traffic using two packet traces: the "Abilene-I" trace collected on the Abilene network [4] and a one day long trace from the "2009-M57 Patents" dataset [3] collected in a small-enterprise 1Gbps network. The former contains only packet headers while the latter includes packet payloads as well.

Our generator software is composed of many traffic sources, each reading from a copy of the original traces. By mixing multiple traffic sources we can generate high bit rates as well as introduce burstiness and sudden spikes in the traffic load. This helps in testing the responsiveness and stability of our power saving algorithm.

We are interested in evaluating the performance of our system with a broad range of applications. To this end, we run experiments with the following packet processing applications: IPv4 routing, a Netflow-like monitoring application that maintains per-flow state, an IPSEC implementation that encrypts every packet using AES-128 and the Redundancy Elimination implementation from [10] which removes duplicate content from the traffic. Note that we use all applications *as-is*, without modifications from their original Click-based implementations.

This set of applications is quite representative of typical networking applications. Routing is a state-less application where each packet can be processed independently of the other. Netflow is stateful and requires to create and maintain a large amount of per-flow state. IPSEC is very CPU intensive but stateless. Finally, Redundancy Elimination is both stateful (keeps information about previously observed packets) and CPU intensive (to find duplicate content in packet payloads).

We generate similar input traffic profiles for all applications. Figure 14 shows the traffic load over time for the different applications. We tune the traffic profiles so that the average utilization of the router over the duration of the experiment is around 35%. This results in



(a) Routing

(b) NetFlow

(c) IPSEC

(d) Redundancy Elimination

Figure 15: *Power usage with (a) IPv4 routing, (b) NetFlow, (c) IPSEC, (d) Redundancy Elimination.*

different bit rates for different applications as they differ in per-packet processing cost (with Redundancy Elimination being the most demanding). The load is evenly distributed across the four interfaces and the routing table is uniform so that no output interface has to forward more than the 10 Gbps it can handle.

We first focus on the power consumption of the various applications and then measure the impact of *PowerSave* on more traditional performance metrics such as latency, drops and packet reordering.

Power consumption. Figure 15[a-d] shows the power consumption over time for the four applications under study. In the figures we compare the power consumption of the default Click implementation, the NAPI-Click and Click with PowerSave. As expected, unmodified Click fares the worst with a power consumption hovering around 270 W for all applications at all traffic rates. The power usage with NAPI-Click and PowerSave instead tracks, to varying degree, the input traffic rate.

PowerSave is consistently the most energy efficient at all rates for all applications. The advantage of PowerSave is more prominent when the load is fairly low $(10-20\%)$ as that is when consolidating the work across cores yields the maximum benefits. Over the duration

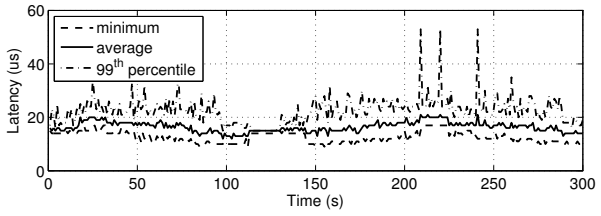


Figure 16: *Average, minimum and 99th percentile of the packet forwarding latency when using Click with the power save algorithm.*

of the experiments, PowerSave yields an overall energy saving between 12% and 25% compared to NAPI-Click and 27-50% compared to unmodified Click.

Traditional performance metrics. We turn our attention to other metrics beyond power consumption, namely: packet loss, latency and reordering. Our algorithm may introduce loss or high latency by turning on cores too slowly in the face of varying traffic demands. Reordering may be introduced when packets belonging to the same flow are redirected to a different core.

Regarding packet losses, the PowerSave algorithm reacts quickly enough to avoid packet drops. Indeed, no packet losses were observed in any of our experiments.

We also measured the latency of packets traversing the router. Figure 16 plots the average, minimum and 99th percentile over each 1s interval – we collect a latency sample every 1ms. In the interest of space we only plot results for one PowerSave experiment with IPv4 Routing. Other experiments show similar behavior [29].

The average latency hovers in the $15-20$ μs range. The same experiments with Click unmodified yield an average latency of 11 μs. The 99th percentile of the latency is up to 4 times the average latency – it peaks at 55 μs. Some of the peaks are unavoidable since waking up a core on from C6 may introduce a latency of up to 85 μs. However, that is limited to the first packet that reaches the queue handled by that core.

As a last performance metric we also measured packet reordering. In our system, reordering can only occur when traffic is diverted to a new queue. Hence, two back to back packets, A and B, belonging to the same flow may be split across two queues and incur very different latency. Given that new queues are activated only when the current ones start to fill up, it is possible to have packet A at the back of one queue while packet B is first in line to be served on the new queue. On the other hand, packets in the newly activated queue will not be processed until the core assigned to it exits a C6 state. Given that in our setting the adaptation algorithm aims for $\approx$ 10 μs of queueing delay, it is quite likely that packet A will be served while the new core is still exiting C6. We confirmed this conjecture in our experiments

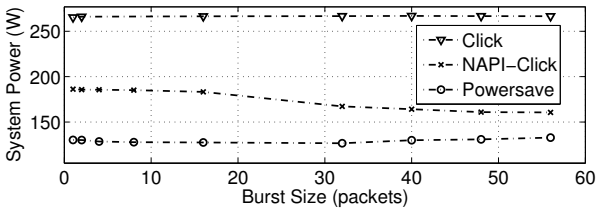


Figure 17: *System power varying the burstiness of the input traffic. The average traffic rate is constant at* 2.5 *Gbps per interface*

where we have measured zero packets being reordered.

Impact of traffic burstiness. Our results so far were based on input traffic whose fine-grained "burstiness" derived from the natural variability associated with a software-based traffic source and multiplexing traffic from multiple such sources. To study the impact of fine-grained burstiness in a more controlled manner, we modified the traffic sources to generate traffic following an on/off pattern. During an "on" period the traffic source generates a burst of back-to-back packets at the maximum rate. We control the burstiness by setting the length of the "on" period (*i.e.*, the number of back-to-back packets) while keeping the average packet rate constant.

Fig. 17 shows the power usage of our system as we vary the burstiness of the sources (a value of 1 corresponds to uniform traffic). The average traffic rate is around 2.5 Gbps per interface. In the interest of space we only plot the results for IPv4 Routing. Other applications exhibit a similar behavior [29]. As we can see the power usage varies very little as we increase the burstiness. This shows that the controller is stable enough to make sure short-term burstiness does not impact its performance. As expected, the power usage with unmodified Click is flat while NAPI-Click benefits from traffic burstiness as interrupts are spaced out more with burstier traffic. However, for all levels of burstiness, PowerSave is clearly the most energy efficient. We measured also latency and packet loss and saw similar results where burstiness has little or no impact. We refer the reader to [29] for a detailed analysis of the latency results.

8 Conclusion

We tackle the problem of improving the power efficiency of network routers without compromising their performance. We studied the power-*vs*-performance tradeoffs in the context of commodity server hardware and derived three guiding principles on how to combine different power management options. Based on these guidelines we build a prototype Click-based software router whose power consumption grows in proportion to the offered load, using between 50% and 100% of the original power, without compromising on peak performance.

References

[1] BlueCoat Proxy. `http://www.bluecoat.com`.

[2] Cisco Green Research Symposium. `http://goo.gl/MOUNl`.

[3] "M57 Patents" Dataset. `http://goo.gl/JGgw4`.

[4] NLANR: Internet Measurement and Analysis. `http://moat.nlanr.net`.

[5] Receive-Side Scaling Enhancements in Windows Server 2008. `http://goo.gl/hsiU9`.

[6] Riverbed Steelhead:. `http://www.riverbed.com`.

[7] SAMSUNG Develops Industry's First DDR4DRAM, Using 30nm Class Technology. `http://goo.gl/8aws`.

[8] Vyatta Router and Firewall. `http://www.vyatta.com`.

[9] Media Access Control Parameters, Physical Layers and Management Parameters for Energy-Efficient Ethernet. IEEE 802.3az Standard, 2010.

[10] A. Anand et al. Packet caches on routers-the implications of universal redundant traffic elimination. In *ACM Sigcomm*, 2008.

[11] G. Ananthanarayanan and R. Katz. Greening the switch. In *HotPower*, 2008.

[12] M. Anwer et al. Switchblade: A platform for rapid deployment of network protocols on programmable hardware. 2010.

[13] J. S. Chase et al. Managing energy and server resources in hosting centers. In *SOSP*, 2001.

[14] M. Dobrescu et al. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *ACM SOSP*, 2009.

[15] W. Fisher et al. Greening Backbone Networks: Reducing Energy Consumption by Shutting Off Cables in Bundled Links. In *ACM Green Networking Workshop*, 2010.

[16] M. Gupta and S. Singh. Greening of the Internet. In *ACM Sigcomm*, 2003.

[17] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated Software Router. In *ACM Sigcomm*, 2010.

[18] B. Heller et al. ElasticTree: Saving Energy in Data Center Networks. In *ACM NSDI*, 2010.

[19] V. Janapa Reddi et al. Web search using mobile cores: quantifying and mitigating the price of efficiency. In *ISCA*, 2010.

[20] C. Joseph et al. Power awareness in network design and routing. In *IEEE Infocom*, 2008.

[21] E. Kohler, , et al. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[22] K. Lim et al. Understanding and designing new server architectures for emerging warehouse-computing. In *ISCA*, 2008.

[23] G. Lu et al. Serverswitch: A programmable and high performance platform for data center networks. In *NSDI*, 2011.

[24] P. Mahadevan et al. A Power Benchmarking Framework for Network Devices. In *IFIP Networking*, 2009.

[25] G. Maier et al. On Dominant Characteristics of Residential Broadband Internet Traffic. In *ACM IMC*, 2009.

[26] D. Meisner et al. Power management of online data-intensive services. In *ISCA, June*, 2011.

[27] D. Meisner, B. T. Gold, and T. F. Wenisch. Powernap: eliminating server idle power. In *ASPLOS*, 2009.

[28] S. Nedevschi et al. Reducing Network Energy Consumption via Sleeping and Rate-Adaptation. In *Proc. of NSDI*, 2008.

[29] L. Niccolini et al. Building a Power-Proportional Router (extended version). Technical report, UCB, 2011.

[30] P. Reviriego et al. Using coordinated transmission with energy efficient ethernet. In *Networking*, May 2011.

[31] V. Sekar et al. The Middlebox Manifesto: Enabling Innovation in Middlebox Deployment. In *ACM HotNets-X*, 2011.

[32] SPECpower ssj2008 results. `http://goo.gl/h6Z2F`.

[33] N. Tolia et al. Delivering energy proportionality with non energy-proportional systems. In *HotPower*, 2008.

[34] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah. Analyzing the energy efficiency of a database server. In *SIGMOD*, 2010.

netmap: a novel framework for fast packet I/O

Luigi Rizzo[*], *Università di Pisa, Italy*

Abstract

Many applications (routers, traffic monitors, firewalls, etc.) need to send and receive packets at line rate even on very fast links. In this paper we present *netmap*, a novel framework that enables commodity operating systems to handle the millions of packets per seconds traversing 1..10 Gbit/s links, without requiring custom hardware or changes to applications.

In building *netmap*, we identified and successfully reduced or removed three main packet processing costs: per-packet dynamic memory allocations, removed by preallocating resources; system call overheads, amortized over large batches; and memory copies, eliminated by sharing buffers and metadata between kernel and userspace, while still protecting access to device registers and other kernel memory areas. Separately, some of these techniques have been used in the past. The novelty in our proposal is not only that we exceed the performance of most of previous work, but also that we provide an architecture that is tightly integrated with existing operating system primitives, not tied to specific hardware, and easy to use and maintain.

netmap has been implemented in FreeBSD and Linux for several 1 and 10 Gbit/s network adapters. In our prototype, a single core running at 900 MHz can send or receive 14.88 Mpps (the peak packet rate on 10 Gbit/s links). This is more than 20 times faster than conventional APIs. Large speedups (5x and more) are also achieved on user-space Click and other packet forwarding applications using a libpcap emulation library running on top of *netmap*.

1 Introduction

General purpose OSes provide a rich and flexible environment for running, among others, many packet processing and network monitoring and testing tasks. The high rate raw packet I/O required by these applications is not the intended target of general purpose OSes. Raw sockets, the Berkeley Packet Filter [14] (BPF), the AF_SOCKET family, and equivalent APIs have been used to build all sorts of network monitors, traffic generators, and generic routing systems. Performance, however, is inadequate for the millions of packets per second (*pps*) that can be present on 1..10 Gbit/s links. In search of better performance, some systems (see Section 3) either run completely in the kernel, or bypass the device driver and the entire network stack by exposing the NIC's data structures to user space applications. Efficient as they may be, many of these approaches depend on specific hardware features, give unprotected access to hardware, or are poorly integrated with the existing OS primitives.

The *netmap* framework presented in this paper combines and extends some of the ideas presented in the past trying to address their shortcomings. Besides giving huge speed improvements, *netmap* does not depend on specific hardware[1], has been fully integrated in FreeBSD and Linux with minimal modifications, and supports unmodified libpcap clients through a compatibility library.

One metric to evaluate our framework is performance: in our implementation, moving one packet between the wire and the userspace application has an amortized cost of less than 70 CPU clock cycles, which is at least one order of magnitude faster than standard APIs. In other words, a single core running at 900 MHz can source or sink the 14.88 Mpps achievable on a 10 Gbit/s link. The same core running at 150 MHz is well above the capacity of a 1 Gbit/s link.

Other, equally important, metrics are safety of operation and ease of use. *netmap* clients cannot possibly crash the system, because device registers and critical kernel memory regions are not exposed to clients,

[*]This work was funded by the EU FP7 project CHANGE (257422).

[1] *netmap* can give isolation even without hardware mechanisms such as IOMMU or VMDq, and is orthogonal to hardware offloading and virtualization mechanisms (checksum, TSO, LRO, VMDc, etc.)

and they cannot inject bogus memory pointers in the kernel (these are often vulnerabilities of other schemes based on shared memory). At the same time, *netmap* uses an extremely simple data model well suited to zero-copy packet forwarding; supports multi-queue adapters; and uses standard system calls (`select()/poll()`) for event notification. All this makes it very easy to port existing applications to the new mechanism, and to write new ones that make effective use of the *netmap* API.

In this paper we will focus on the architecture and features of *netmap*, and on its core performance. In a related Infocom paper [19] we address a different problem: (how) can applications make good use of a fast I/O subsystem such as netmap ? [19] shows that significant performance bottlenecks may emerge in the applications themselves, although in some cases we can remove them and make good use of the new infrastructure.

In the rest of this paper, Section 2 gives some background on current network stack architecture and performance. Section 3 presents related work, illustrating some of the techniques that *netmap* integrates and extends. Section 4 describes *netmap* in detail. Performance data are presented in Section 5. Finally, Section 6 discusses open issues and our plans for future work.

2 Background

There has always been interest in using general purpose hardware and Operating Systems to run applications such as software switches [15], routers [6, 4, 5], firewalls, traffic monitors, intrusion detection systems, or traffic generators. While providing a convenient development and runtime environment, such OSes normally do not offer efficient mechanisms to access raw packet data at high packet rates. This Section illustrates the organization of the network stack in general purpose OSes and shows the processing costs of the various stages.

2.1 NIC data structures and operation

Network adapters (NICs) normally manage incoming and outgoing packets through circular queues (*rings*) of buffer descriptors, as in Figure 1. Each slot in the ring contains the length and physical address of the buffer. CPU-accessible registers in the NIC indicate the portion of the ring available for transmission or reception.

On reception, incoming packets are stored in the next available buffer (possibly split in multiple fragments), and length/status information is written back to the slot to indicate the availability of new data. Interrupts notify the CPU of these events. On the transmit side, the NIC expects the OS to fill buffers with data to be sent. The request to send new packets is issued by writing into the

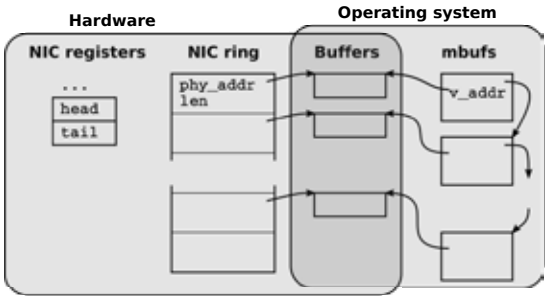


Figure 1: Typical NIC's data structures and their relation with the OS data structures.

registers of the NIC, which in turn starts sending packets marked as available in the TX ring.

At high packet rates, interrupt processing can be expensive and possibly lead to the so-called "receive livelock" [16], or inability to perform any useful work above a certain load. Polling device drivers [10, 16, 18] and the hardware interrupt mitigation implemented in recent NICs solve this problem.

Some high speed NICs support multiple transmit and receive rings. This helps spreading the load on multiple CPU cores, eases on-NIC traffic filtering, and helps decoupling virtual machines sharing the same hardware.

2.2 Kernel and user APIs

The OS maintains shadow copies of the NIC's data structures. Buffers are linked to OS-specific, device-independent containers (`mbufs` [22] or equivalent structures such as `sk_buffs` and `NdisPackets`). These containers include large amounts of metadata about each packet: size, source or destination interface, and attributes and flags to indicate how the buffers should be processed by the NIC and the OS.

Driver/OS: The software interface between device drivers and the OS usually assumes that packets, in both directions, can be split into an arbitrary number of fragments; both the device drivers and the host stack must be prepared to handle the fragmentation. The same API also expects that subsystems may retain packets for deferred processing, hence buffers and metadata cannot be simply passed by reference during function calls, but they must be copied or reference-counted. This flexibility is paid with a significant overhead at runtime.

These API contracts, perhaps appropriate 20-30 years ago when they were designed, are far too expensive for today's systems. The cost of allocating, managing and navigating through buffer chains often exceeds that of linearizing their content, even when producers do indeed generate fragmented packets (e.g. TCP when prepending headers to data from the socket buffers).

Raw packet I/O: The standard APIs to read/write raw packets for user programs require at least one memory copy to move data and metadata between kernel and user space, and one system call per packet (or, in the best cases, per batch of packets). Typical approaches involve opening a socket or a Berkeley Packet Filter [14] device, and doing I/O through it using `send()/recv()` or specialized `ioctl()` functions.

2.3 Case study: FreeBSD sendto()

To evaluate how time is spent in the processing of a packet, we have instrumented the `sendto()` system call in FreeBSD[2] so that we can force an early return from the system call at different depths, and estimate the time spent in the various layers of the network stack. Figure 2 shows the results when a test program loops around a `sendto()` on a bound UDP socket. In the table, "time" is the average time per packet when the return point is at the beginning of the function listed on the row; "delta" is the difference between adjacent rows, and indicates the time spent at each stage of the processing chain. As an example, the userspace code takes 8 ns per iteration, entering the kernel consumes an extra 96 ns, and so on.

As we can see, we find several functions at all levels in the stack consuming a significant share of the total execution time. Any network I/O (be it through a TCP or raw socket, or a BPF writer) has to go through several expensive layers. Of course we cannot avoid the system call; the initial mbuf construction/data copy is expensive, and so are the route and header setup, and (surprisingly) the MAC header setup. Finally, it takes a long time to translate mbufs and metadata into the NIC format. Local optimizations (e.g. caching routes and headers instead of rebuilding them every time) can give modest improvements, but we need radical changes at all layers to gain the tenfold speedup necessary to work at line rate on 10 Gbit/s interfaces.

What we show in this paper is how fast can we become if we take such a radical approach, while still enforcing safety checks on user supplied data through a system call, and providing a libpcap-compatible API.

File	Function/description	time ns	delta ns
user program	`sendto` system call	8	96
uipc_syscalls.c	`sys_sendto`	104	
uipc_syscalls.c	`sendit`	111	
uipc_syscalls.c	`kern_sendit`	118	
uipc_socket.c	`sosend`	—	
uipc_socket.c	`sosend_dgram` sockbuf locking, mbuf allocation, copyin	146	137
udp_usrreq.c	`udp_send`	273	
udp_usrreq.c	`udp_output`	273	57
ip_output.c	`ip_output` route lookup, ip header setup	330	198
if_ethersubr.c	`ether_output` MAC header lookup and copy, loopback	528	162
if_ethersubr.c	`ether_output_frame`	690	
ixgbe.c	`ixgbe_mq_start`	698	
ixgbe.c	`ixgbe_mq_start_locked`	720	
ixgbe.c	`ixgbe_xmit` mbuf mangling, device programming	730	220
–	on wire	950	

Figure 2: The path and execution times for `sendto()` on a recent FreeBSD HEAD 64-bit, i7-870 at 2.93 GHz + Turboboost, Intel 10 Gbit NIC and ixgbe driver. Measurements done with a single process issuing sendto() calls. Values have a 5% tolerance and are averaged over multiple 5s tests.

3 Related (and unrelated) work

It is useful at this point to present some techniques proposed in the literature, or used in commercial systems, to improve packet processing speeds. This will be instrumental in understanding their advantages and limitations, and to show how our framework can use them.

Socket APIs: The Berkeley Packet Filter, or BPF [14], is one of the most popular systems for direct access to raw packet data. BPF taps into the data path of a network device driver, and dispatches a copy of each sent or received packet to a file descriptor, from which userspace processes can read or write. Linux has a similar mechanism through the AF_PACKET socket family. BPF can coexist with regular traffic from/to the system, although usually BPF clients put the card in promiscuous mode, causing large amounts of traffic to be delivered to the host stack (and immediately dropped).

Packet filter hooks: Netgraph (FreeBSD), Netfilter (Linux), and Ndis Miniport drivers (Windows) are in-kernel mechanisms used when packet duplication is not necessary, and instead the application (e.g. a firewall) must be interposed in the packet processing chain. These hooks intercept traffic from/to the driver and pass it to processing modules without additional data copies. The packet filter hooks rely on the standard mbuf/sk_buff based packet representation.

Direct buffer access: One easy way to remove the data copies involved in the kernel-userland transition

[2] We expect similar numbers on Linux and Windows.

is to run the application code directly within the kernel. Kernel-mode Click [10] supports this approach [4]. Click permits an easy construction of packet processing chains through the composition of modules, some of which support fast access to the NIC (even though they retain an sk_buff-based packet representation).

The kernel environment is however very constrained and fragile, so a better choice is to expose packet buffers to userspace. Examples include PF_RING [2] and Linux PACKET_MMAP, which export to userspace clients a shared memory region containing multiple pre-allocated packet buffers. The kernel is in charge of copying data between sk_buffs and the shared buffers, so that no custom device drivers are needed. This amortizes the system call costs over batches of packets, but retains the data copy and sk_buff management overhead. Possibly (but we do not have detailed documentation) this is also how the "Windows Registered I/O API" (RIO) [20] works.

Better performance can be achieved by running the full stack, down to NIC access, in userspace. This requires custom device drivers, and poses some risks because the NIC's DMA engine can write to arbitrary memory addresses (unless limited by hardware mechanisms such as IOMMUs), so a misbehaving client can potentially trash data anywhere in the system. Examples in this category include UIO-IXGBE [11], PF_RING-DNA [3], and commercial solutions including Intel's DPDK [8] and SolarFlare's OpenOnload [21].

Van Jacobson's NetChannels [9] have some similarities to our work, at least on the techniques used to accelerate performance: remove sk_bufs, avoid packet processing in interrupt handlers, and map buffers to userspace where suitable libraries implement the whole protocol processing. The only documentation available [9] shows interesting speedups, though subsequent attempts to implement the same ideas in Linux (see [13]) were considered unsatisfactory, presumably because of additional constraints introduced trying to remain 100% compatible with the existing kernel network architecture.

The PacketShader [5] I/O engine (PSIOE) is another close relative to our proposal, especially in terms of performance. PSIOE uses a custom device driver that replaces the sk_buff-based API with a simpler one, using preallocated buffers. Custom `ioctl()`s are used to synchronize the kernel with userspace applications, and multiple packets are passed up and down through a memory area shared between the kernel and the application. The kernel is in charge of copying packet data between the shared memory and packet buffers. Unlike *netmap*, PSIOE only supports one specific NIC, and does not support `select()/poll()`, requiring modifications to applications in order to let them use the new API.

Hardware solutions: Some hardware has been designed specifically to support high speed packet capture, or possibly generation, together with special features such as timestamping, filtering, forwarding. Usually these cards come with custom device drivers and user libraries to access the hardware. As an example, DAG [1, 7] cards are FPGA-based devices for wire-rate packet capture and precise timestamping, using fast onboard memory for the capture buffers (at the time they were introduced, typical I/O buses were unable to sustain line rate at 1 and 10 Gbit/s). NetFPGA [12] is another example of an FPGA-based card where the firmware of the card can be programmed to implement specific functions directly in the NIC, offloading some work from the CPU.

3.1 Unrelated work

A lot of commercial interest, in high speed networking, goes to TCP acceleration and hardware virtualization, so it is important to clarify where *netmap* stands in this respect. *netmap* **is a framework to reduce the cost of moving traffic between the hardware and the host stack**. Popular hardware features related to TCP acceleration, such as hardware checksumming or even encryption, Tx Segmentation Offloading, Large Receive Offloading, are completely orthogonal to our proposal: they reduce some processing in the host stack but do not address the communication with the device. Similarly orthogonal are the features relates to virtualization, such as support for multiple hardware queues and the ability to assign traffic to specific queues (VMDq) and/or queues to specific virtual machines (VMDc, SR-IOV). We expect to run *netmap* within virtual machines, although it might be worthwhile (but not the focus of this paper) to explore how the ideas used in *netmap* could be used within a hypervisor to help the virtualization of network hardware.

4 Netmap

The previous survey shows that most related proposals have identified, and tried to remove, the following high cost operations in packet processing: data copying, metadata management, and system call overhead.

Our framework, called *netmap*, is a system to give user space applications very fast access to network packets, both on the receive and the transmit side, and including those from/to the host stack. Efficiency does not come at the expense of safety of operation: potentially dangerous actions such as programming the NIC are validated by the OS, which also enforces memory protection. Also, a distinctive feature of *netmap* is the attempt to design and implement an API that is simple to use, tightly integrated with existing OS mechanisms, and not tied to a specific device or hardware features.

netmap achieves its high performance through several techniques:

- a lightweight metadata representation which is compact, easy to use, and hides device-specific features. Also, the representation supports processing of large number of packets in each system call, thus amortizing its cost;
- linear, fixed size packet buffers that are preallocated when the device is opened, thus saving the cost of per-packet allocations and deallocations;
- removal of data-copy costs by granting applications direct, protected access to the packet buffers. The same mechanism also supports zero-copy transfer of packets between interfaces;
- support of useful hardware features (such as multiple hardware queues).

Overall, we use each part of the system for the task it is best suited to: the NIC to move data quickly between the network and memory, and the OS to enforce protection and provide support for synchronization.

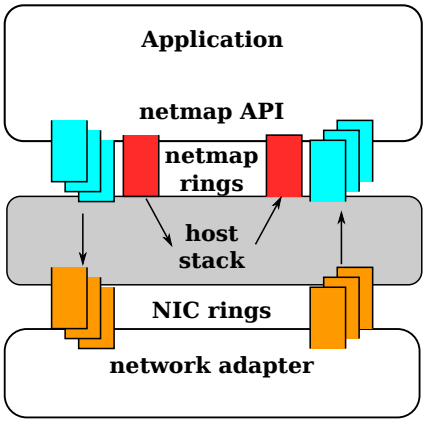


Figure 3: In netmap mode, the NIC rings are disconnected from the host network stack, and exchange packets through the netmap API. Two additional netmap rings let the application talk to the host stack.

At a very high level, when a program requests to put an interface in *netmap* mode, the NIC is partially disconnected (see Figure 3) from the host protocol stack. The program gains the ability to exchange packets with the NIC and (separately) with the host stack, through circular queues of buffers (*netmap rings*) implemented in shared memory. Traditional OS primitives such as `select()/poll()` are used for synchronization. Apart from the disconnection in the data path, the operating system is unaware of the change so it still continues to use and manage the interface as during regular operation.

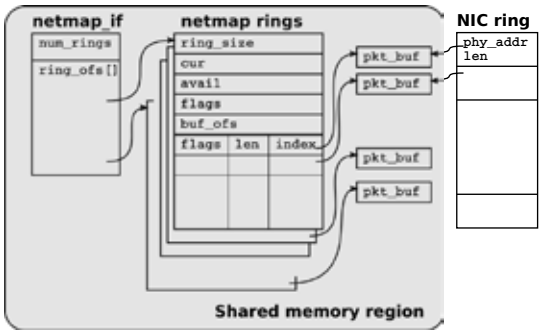


Figure 4: User view of the shared memory area exported by netmap.

4.1 Data structures

The key component in the *netmap* architecture are the data structures shown in Figure 4. They are designed to provide the following features: 1) reduced/amortized per-packet overheads; 2) efficient forwarding between interfaces; 3) efficient communication between the NIC and the host stack; and 4) support for multi-queue adapters and multi core systems.

netmap supports these features by associating to each interface three types of user-visible objects, shown in Figure 4: *packet buffers*, *netmap rings*, and *netmap_if* descriptors. All objects for all netmap-enabled interfaces in the system reside in the same memory region, allocated by the kernel in a non-pageable area, and shared by all user processes. The use of a single region is convenient to support zero-copy forwarding between interfaces, but it is trivial to modify the code so that different interfaces or groups of interfaces use separate memory regions, gaining better isolation between clients.

Since the shared memory is mapped by processes and kernel threads in different virtual address spaces, any memory reference contained in that region must use *relative* addresses, so that pointers can be calculated in a position-independent way. The solution to this problem is to implement references as offsets between the parent and child data structures.

Packet buffers have a fixed size (2 Kbytes in the current implementation) and are shared by the NICs and user processes. Each buffer is identified by a unique *index*, that can be easily translated into a virtual address by user processes or by the kernel, and into a physical address used by the NIC's DMA engines. Buffers for all netmap rings are preallocated when the interface is put into netmap mode, so that during network I/O there is never the need to allocate them. The metadata describing the buffer (index, data length, some flags) are stored into *slots* that are part of the netmap rings described next. Each buffer is referenced by a netmap ring and by the

corresponding hardware ring.

A *netmap ring* is a device-independent replica of the circular queue implemented by the NIC, and includes:

- `ring_size`, the number of slots in the ring;
- `cur`, the current read or write position in the ring;
- `avail`, the number of available buffers (received packets in RX rings, empty slots in TX rings);
- `buf_ofs`, the offset between the ring and the beginning of the array of (fixed-size) packet buffers;
- `slots[]`, an array with `ring_size` entries. Each slot contains the index of the corresponding packet buffer, the length of the packet, and some flags used to request special operations on the buffer.

Finally, a *netmap_if* contains read-only information describing the interface, such as the number of rings and an array with the memory offsets between the `netmap_if` and each netmap ring associated to the interface (once again, offsets are used to make addressing position-independent).

4.1.1 Data ownership and access rules

The *netmap* data structures are shared between the kernel and userspace, but the ownership of the various data areas is well defined, so that there are no races. In particular, the *netmap_ring* is always owned by the userspace application except during the execution of a system call, when it is updated by the kernel code still in the context of the user process. Interrupt handlers and other kernel threads never touch a netmap ring.

Packet buffers between `cur` and `cur+avail-1` are owned by the userspace application, whereas the remaining buffers are owned by the kernel (actually, only the NIC accesses these buffers). The boundary between these two regions is updated during system calls.

4.2 The netmap API

Programs put an interface in *netmap* mode by opening the special device `/dev/netmap` and issuing an

```
ioctl(.., NIOCREG, arg)
```

on the file descriptor. The argument contains the interface name, and optionally the indication of which rings we want to control through this file descriptor (see Section 4.2.2). On success, the function returns the size of the shared memory region where all data structures are located, and the offset of the `netmap_if` within the region. A subsequent `mmap()` on the file descriptor makes the memory accessible in the process' address space.

Once the file descriptor is bound to one interface and its ring(s), two more `ioctl()`s support the transmission and reception of packets. In particular, transmissions require the program to fill up to `avail` buffers in the TX ring, starting from slot `cur` (packet lengths are written to the `len` field of the slot), and then issue an

```
ioctl(.., NIOCTXSYNC)
```

to tell the OS about the new packets to send. This system call passes the information to the kernel, and on return it updates the `avail` field in the netmap ring, reporting slots that have become available due to the completion of previous transmissions.

On the receive side, programs should first issue an

```
ioctl(.., NIOCRXSYNC)
```

to ask the OS how many packets are available for reading; then their lengths and payloads are immediately available through the slots (starting from `cur`) in the netmap ring.

Both NIOC*SYNC `ioctl()`s are non blocking, involve no data copying (except from the synchronization of the slots in the netmap and hardware rings), and can deal with multiple packets at once. These features are essential to reduce the per-packet overhead to very small values. The in-kernel part of these system calls does the following:

- validates the `cur/avail` fields and the content of the slots involved (lengths and buffer indexes, both in the netmap and hardware rings);
- synchronizes the content of the slots between the netmap and the hardware rings, and issues commands to the NIC to advertise new packets to send or newly available receive buffers;
- updates the `avail` field in the netmap ring.

The amount of work in the kernel is minimal, and the checks performed make sure that the user-supplied data in the shared data structure do not cause system crashes.

4.2.1 Blocking primitives

Blocking I/O is supported through the `select()` and `poll()` system calls. Netmap file descriptors can be passed to these functions, and are reported as ready (waking up the caller) when `avail` > 0. Before returning from a `select()/poll()`, the system updates the status of the rings, same as in the NIOC*SYNC ioctls. This way, applications spinning on an eventloop require only one system call per iteration.

4.2.2 Multi-queue interfaces

For cards with multiple ring pairs, file descriptors (and the related `ioctl()` and `poll()`) can be configured in one of two modes, chosen through the `ring_id` field in

the argument of the NIOCREG `ioctl()`. In the default mode, the file descriptor controls all rings, causing the kernel to check for available buffers on any of them. In the alternate mode, a file descriptor is associated to a single TX/RX ring pair. This way multiple threads/processes can create separate file descriptors, bind them to different ring pairs, and operate independently on the card without interference or need for synchronization. Binding a thread to a specific core just requires a standard OS system call, `setaffinity()`, without the need of any new mechanism.

4.2.3 Example of use

The example below (the core of the packet generator used in Section 5) shows the simplicity of use of the *netmap* API. Apart from a few macros used to navigate through the data structures in the shared memory region, netmap clients do not need any library to use the system, and the code is extremely compact and readable.

```
fds.fd = open("/dev/netmap", O_RDWR);
strcpy(nmr.nm_name, "ix0");
ioctl(fds.fd, NIOCREG, &nmr);
p = mmap(0, nmr.memsize, fds.fd);
nifp = NETMAP_IF(p, nmr.offset);
fds.events = POLLOUT;
for (;;) {
  poll(fds, 1, -1);
  for (r = 0; r < nmr.num_queues; r++) {
    ring = NETMAP_TXRING(nifp, r);
    while (ring->avail-- > 0) {
      i = ring->cur;
      buf = NETMAP_BUF(ring, ring->slot[i].buf_index);
      ... store the payload into buf ...
      ring->slot[i].len =  ... // set packet length
      ring->cur = NETMAP_NEXT(ring, i);
    }
  }
}
```

4.3 Talking to the host stack

Even in netmap mode, the network stack in the OS is still in charge of controlling the interface (through ifconfig and other functions), and will generate (and expect) traffic to/from the interface. This traffic is handled with an additional pair of netmap rings, which can be bound to a netmap file descriptor with a NIOCREG call.

An NIOCTXSYNC on one of these rings encapsulates buffers into mbufs and then passes them to the host stack, as if they were coming from the physical interface. Packets coming from the host stack instead are queued to the "host stack" netmap ring, and made available to the netmap client on subsequent NIOCRXSYNCs.

It is then a responsibility of the netmap client to make sure that packets are properly passed between the rings connected to the host stack and those connected to the NIC. Implementing this feature is straightforward, possibly even using the zero-copy technique shown in Section 4.5. This is also an ideal opportunity to implement functions such as firewalls, traffic shapers and NAT boxes, which are normally attached to packet filter hooks.

4.4 Safety considerations

The sharing of memory between the kernel and the multiple user processes who can open `/dev/netmap` poses the question of what safety implications exist in the usage of the framework.

Processes using *netmap*, even if misbehaving, *cannot cause the kernel to crash*, unlike many other high-performance packet I/O systems (e.g. UIO-IXGBE, PF_RING-DNA, in-kernel Click). In fact, the shared memory area does not contain critical kernel memory regions, and buffer indexes and lengths are always validated by the kernel before being used.

A misbehaving process can however corrupt someone else's netmap rings or packet buffers. The easy cure for this problem is to implement a separate memory region for each ring, so clients cannot interfere. This is straightforward in case of hardware multiqueues, or it can be trivially simulated in software without data copies. These solutions will be explored in future work.

4.5 Zero-copy packet forwarding

Having all buffers for all interfaces in the same memory region, zero-copy packet forwarding between interfaces only requires to swap the buffers indexes between the receive slot on the incoming interface and the transmit slot on the outgoing interface, and update the length and flags fields accordingly:

```
...
src = &src_nifp->slot[i]; /* locate src and dst slots */
dst = &dst_nifp->slot[j];
/* swap the buffers */
tmp = dst->buf_index;
dst->buf_index = src->buf_index;
src->buf_index = tmp;
/* update length and flags */
dst->len = src->len;
/* tell kernel to update addresses in the NIC rings */
dst->flags = src->flags = BUF_CHANGED;
...
```

The swap enqueues the packet on the output interface, and at the same time refills the input ring with an empty buffer without the need to involve the memory allocator.

4.6 libpcap compatibility

An API is worth little if there are no applications that use it, and a significant obstacle to the deployment of new APIs is the need to adapt existing code to them.

Following a common approach to address compatibility problems, one of the first things we wrote on top

of netmap was a small library that maps `libpcap` calls into *netmap* calls. The task was heavily simplified by the fact that *netmap* uses standard synchronization primitives, so we just needed to map the read/write functions (`pcap_dispatch()/pcap_inject()`) into equivalent netmap calls – about 20 lines of code in total.

4.7 Implementation

In the design and development of *netmap*, a fair amount of work has been put into making the system maintainable and performant. The current version, included in FreeBSD, consists of about 2000 lines of code for system call (ioctl, select/poll) and driver support. There is no need for a userspace library: a small C header (200 lines) defines all the structures, prototypes and macros used by *netmap* clients. We have recently completed a Linux version, which uses the same code plus a small wrapper to map certain FreeBSD kernel functions into their Linux equivalents.

To keep device drivers modifications small (a must, if we want the API to be implemented on new hardware), most of the functionalities are implemented in common code, and each driver only needs to implement two functions for the core of the NIOC*SYNC routines, one function to reinitialize the rings in netmap mode, and one function to export device driver locks to the common code. This reduces individual driver changes, mostly mechanical, to about 500 lines each, (a typical device driver has 4k .. 10k lines of code). *netmap* support is currently available for the Intel 10 Gbit/s adapters (ixgbe driver), and for various 1 Gbit/s adapters (Intel, RealTek, nvidia).

In the *netmap* architecture, device drivers do most of their work (which boils down to synchronizing the NIC and netmap rings) in the context of the userspace process, during the execution of a system call. This improves cache locality, simplifies resource management (e.g. binding processes to specific cores), and makes the system more controllable and robust, as we do not need to worry of executing too much code in non-interruptible contexts. We generally modify NIC drivers so that the interrupt service routine does no work except from waking up any sleeping process. This means that interrupt mitigation delays are directly passed to user processes.

Some trivial optimizations also have huge returns in terms of performance. As an example, we don't reclaim transmitted buffers or look for more incoming packets if a system call is invoked with `avail` $>$ 0. This helps applications that unnecessarily invoke system calls on every packet. Two more optimizations (pushing out any packets queued for transmission even if POLLOUT is not specified; and updating a timestamp within the netmap ring before `poll()` returns) reduce from 3 to 1 the number of system calls in each iteration of the typical event loop – once again a significant performance enhancement for certain applications.

To date we have not tried optimizations related to the use of `prefetch` instructions, or data placements to improve cache behaviour.

5 Performance analysis

We discuss the performance of our framework by first analysing its behaviour for simple I/O functions, and then looking at more complex applications running on top of *netmap*. Before presenting our results, it is important to define the test conditions in detail.

5.1 Performance metrics

The processing of a packet involves multiple subsystems: CPU pipelines, caches, memory and I/O buses. Interesting applications are CPU-bound, so we will focus our measurements on the CPU costs. Specifically, we will measure the work (*system costs*) performed to move packets between the application and the network card. This is precisely the task that *netmap* or other packet-I/O APIs are in charge of. We can split these costs in two components:
i) Per-byte costs are the CPU cycles consumed to move data from/to the NIC's buffers (for reading or writing a packet). This component can be equal to zero in some cases: as an example, *netmap* exports NIC buffers to the application, so it has no per-byte system costs. Other APIs, such as the socket API, impose a data copy to move traffic from/to userspace, and this has a per-byte CPU cost that, taking into account the width of memory buses and the ratio between CPU and memory bus clocks, can be in the range of 0.25 to 2 clock cycles per byte.
ii) Per-packet costs have multiple sources. At the very least, the CPU must update a slot in the NIC ring for each packet. Additionally, depending on the software architecture, each packet might require additional work, such as memory allocations, system calls, programming the NIC's registers, updating statistics and the like. In some cases, part of the operations in the second set can be removed or amortized over multiple packets.

Given that in most cases (and certainly this is true for *netmap*) *per-packet* costs are the dominating component, the most challenging situation in terms of system load is when the link is traversed by the smallest possible packets. For this reason, we run most of our tests with 64 byte packets (60+4 CRC).

Of course, in order to exercise the system and measure its performance we need to run some test code, but we want it to be as simple as possible in order to reduce the interference on the measurement. Our initial tests then use two very simple programs that make

application costs almost negligible: a packet generator which streams pre-generated packets, and a packet receiver which just counts incoming packets.

5.2 Test equipment

We have run most of our experiments on systems equipped with an i7-870 4-core CPU at 2.93 GHz (3.2 GHz with turbo-boost), memory running at 1.33 GHz, and a dual port 10 Gbit/s card based on the Intel 82599 NIC. The numbers reported in this paper refer to the netmap version in FreeBSD HEAD/amd64 as of April 2012. Experiments have been run using directly connected cards on two similar systems. Results are highly repeatable (within 2% or less) so we do not report confidence intervals in the tables and graphs.

netmap is extremely efficient so it saturates a 10 Gbit/s interface even at the maximum packet rate, and we need to run the system at reduced clock speeds to determine the performance limits and the effect of code changes. Our systems can be clocked at different frequencies, taken from a discrete set of values. Nominally, most of them are multiples of 150 MHz, but we do not know how precise the clock speeds are, nor the relation between CPU and memory/bus clock speeds.

The transmit speed (in packets per second) has been measured with a packet generator similar to the one in Section 4.2.3. The packet size can be configured at runtime, as well as the number of queues and threads/cores used to send/receive traffic. Packets are prepared in advance so that we can run the tests with close to zero per-byte costs. The test program loops around a poll(), sending at most B packets (*batch size*) per ring at each round. On the receive side we use a similar program, except that this time we poll for read events and only count packets.

5.3 Transmit speed versus clock rate

As a first experiment we ran the generator with variable clock speeds and number of cores, using a large batch size so that the system call cost is almost negligible. By lowering the clock frequency we can determine the point where the system becomes CPU bound, and estimate the (amortized) number of cycles spent for each packet.

Figure 5 show the results using 1..4 cores and an equivalent number of rings, with 64-byte packets. Throughput scales quite well with clock speed, reaching the maximum line rate near 900 MHz with 1 core. This corresponds to 60-65 cycles/packet, a value which is reasonably in line with our expectations. In fact, in this particular test, the per-packet work is limited to validating the content of the slot in the netmap ring and updating the corresponding slot in the NIC ring. The cost of cache misses (which do exist, especially on the NIC ring) is

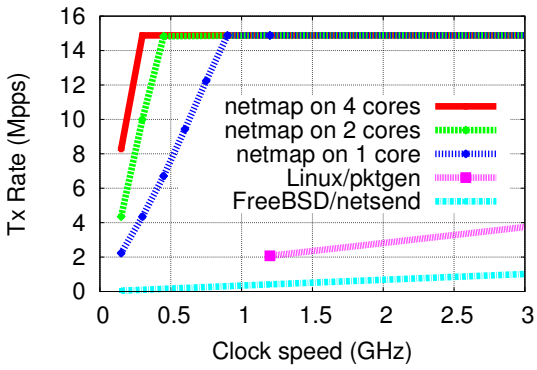


Figure 5: Netmap transmit performance with 64-byte packets, variable clock rates and number of cores, compared to `pktgen` (a specialised, in-kernel generator available on linux, peaking at about 4 Mpps) and a `netsend` (FreeBSD userspace, peaking at 1.05 Mpps).

amortized among all descriptors that fit into a cache line, and other costs (such as reading/writing the NIC's registers) are amortized over the entire batch.

Once the system reaches line rate, increasing the clock speed reduces the total CPU usage because the generator sleeps until an interrupt from the NIC reports the availability of new buffers. The phenomenon is not linear and depends on the duration of the interrupt mitigation intervals. With one core we measured 100% CPU load at 900 MHz, 80% at 1.2 GHz and 55% at full speed.

Scaling with multiple cores is reasonably good, but the numbers are not particularly interesting because there are no significant contention points in this type of experiment, and we only had a small number of operating points (1..4 cores, 150,300, 450 Mhz) before reaching link saturation.

Just for reference, Figure 5 also reports the maximum throughput of two packet generators representative of the performance achievable using standard APIs. The line at the bottom represents `netsend`, a FreeBSD userspace application running on top of a raw socket. `netsend` peaks at 1.05 Mpps at the highest clock speed. Figure 2 details how the 950 ns/pkt are spent.

The other line in the graph is `pktgen`, an in-kernel packet generator available in Linux, which reaches almost 4 Mpps at maximum clock speed, and 2 Mpps at 1.2 GHz (the minimum speed we could set in Linux). Here we do not have a detailed profile of how time is spent, but the similarity of the device drivers and the architecture of the application suggest that most of the cost is in the device driver itself.

The speed vs. clock results for receive are similar to the transmit ones. netmap can do line rate with 1 core at 900 MHz, at least for packet sizes multiple of 64 bytes.

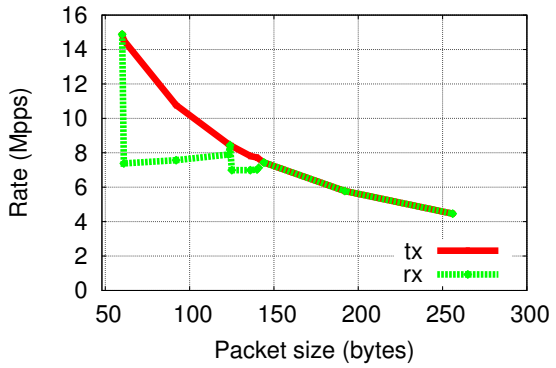


Figure 6: Actual transmit and receive speed with variable packet sizes (excluding Ethernet CRC). The top curve is the transmit rate, the bottom curve is the receive rate. See Section 5.4 for explanations.

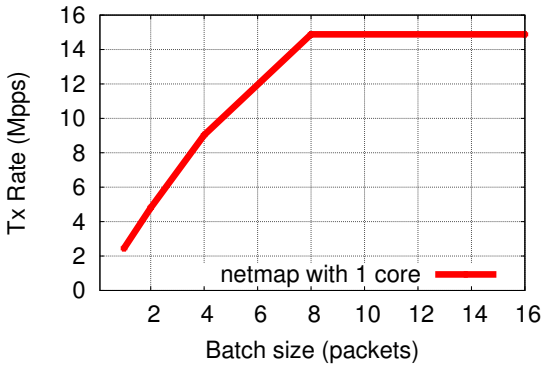


Figure 7: Transmit performance with 1 core, 2.93 GHz, 64-byte packets, and different batch size.

5.4 Speed versus packet size

The previous experiments used minimum-size packets, which is the most critical situation in terms of per-packet overhead. 64-byte packets match very well the bus widths along the various path in the system and this helps the performance of the system. We then checked whether varying the packet size has an impact on the performance of the system, both on the transmit and the receive side.

Transmit speeds with variable packet sizes exhibit the expected $1/size$ behaviour, as shown by the upper curve in Figure 6. The receive side, instead, shows some surprises as indicated by the bottom curve in Figure 6. The maximum rate, irrespective of CPU speed, is achieved only for packet sizes multiples of 64 (or large enough, so that the total data rate is low). At other sizes, receive performance drops (e.g. on Intel CPUs flattens around 7.5 Mpps between 65 and 127 bytes; on AMD CPUs the value is slightly higher). Investigation suggests that the NIC and/or the I/O bridge are issuing read-modify-write cycles for writes that are not a full cache line. Changing the operating mode to remove the CRC from received packets moves the "sweet spots" by 4 bytes (i.e. 64+4, 128+4 etc. achieve line rate, others do not).

We have found that inability to achieve line rate for certain packet size and in transmit or receive mode is present in several NICs, including 1 Gbit/s ones.

5.5 Transmit speed versus batch size

Operating with large batches enhances the throughput of the system as it amortizes the cost of system calls and other potentially expensive operations, such as accessing the NIC's registers. But not all applications have this luxury, and in some cases they are forced to operate in regimes where a system call is issued on each/every few packets. We then ran another set of experiments using different batch sizes and minimum-size packets (64 bytes, including the Ethernet CRC), trying to determine how throughput is affected by the batch size. In this particular test we only used one core, and variable number of queues.

Results are shown in Figure 7: throughput starts at about 2.45 Mpps (408 ns/pkt) with a batch size of 1, and grows quickly with the batch size, reaching line rate (14.88 Mpps) with a batch of 8 packets. The overhead of a standard FreeBSD `poll()` without calling the netmap-specific poll handlers (`netmap_poll()` and `ixgbe_txsync()`) is about 250 ns, so the handling of multiple packets per call is absolutely necessary if we want to reach line rate on 10 Gbit/s and faster interfaces.

5.6 Packet forwarding performance

So far we have measured the costs of moving packets between the wire and the application. This includes the operating systems overhead, but excludes any significant application cost, as well as any data touching operation. It is then interesting to measure the benefit of the *netmap* API when used by more CPU-intensive tasks. Packet forwarding is one of the main applications of packet processing systems, and a good test case for our framework. In fact it involves simultaneous reception and transmission (thus potentially causing memory and bus contention), and may involve some packet processing that consumes CPU cycles, and causes pipeline stalls and cache conflicts. All these phenomena will likely reduce the benefits of using a fast packet I/O mechanism, compared to the simple applications used so far.

We have then explored how a few packet forwarding applications behave when using the new API, either directly or through the `libpcap` compatibility library described in Section 4.6. The test cases are the following:

- `netmap-fwd`, a simple application that forwards packets between interfaces using the zero-copy technique shown in Section 4.5;
- `netmap-fwd + pcap`, as above but using the libpcap emulation instead of the zero-copy code;
- `click-fwd`, a simple Click [10] configuration that passes packets between interfaces:
  ```
  FromDevice(ix0) -> Queue -> ToDevice(ix1)
  FromDevice(ix1) -> Queue -> ToDevice(ix0)
  ```
  The experiment has been run using Click userspace with the system's libpcap, and on top of netmap with the libpcap emulation library;
- `click-etherswitch`, as above but replacing the two queues with an EtherSwitch element;
- `openvswitch`, the OpenvSwitch software with userspace forwarding, both with the system's libpcap and on top of netmap;
- `bsd-bridge`, in-kernel FreeBSD bridging, using the mbuf-based device driver.

Figure 8 reports the measured performance. All experiments have been run on a single core with two 10 Gbit/s interfaces, and maximum clock rate except for the first case where we saturated the link at just 1.733 GHz.

From the experiment we can draw a number of interesting observations:

- native netmap forwarding with no data touching operation easily reaches line rate. This is interesting because it means that full rate bidirectional operation is within reach even for a single core;
- the libpcap emulation library adds a significant overhead to the previous case (7.5 Mpps at full clock vs. 14.88 Mpps at 1.733 GHz means a difference of about 80-100 ns per packet). We have not yet investigated whether/how this can be improved (e.g. by using prefetching);
- replacing the system's libpcap with the netmap-based libpcap emulation gives a speedup between 4 and 8 times for OpenvSwitch and Click, despite the fact that `pcap_inject()` does use data copies. This is also an important result because it means that real-life applications can actually benefit from our API.

5.7 Discussion

In presence of huge performance improvements such as those presented in Figure 5 and Figure 8, which show that *netmap* is 4 to 40 times faster than similar applications using the standard APIs, one might wonder i) how fair is the comparison, and ii) what is the contribution of the various mechanisms to the performance improvement.

Configuration	Mpps
netmap-fwd (1.733 GHz)	14.88
netmap-fwd + pcap	7.50
click-fwd + netmap	3.95
click-etherswitch + netmap	3.10
click-fwd + native pcap	0.49
openvswitch + netmap	3.00
openvswitch + native pcap	0.78
bsd-bridge	0.75

Figure 8: Forwarding performance of our test hardware with various software configurations.

The answer to the first question is that the comparison is indeed fair. All traffic generators in Figure 5 do exactly the same thing and each one tries to do it in the most efficient way, constrained only by the underlying APIs they use. The answer is even more obvious for Figure 8, where in many cases we just use the same unmodified binary on top of two different `libpcap` implementations.

The results measured in different configurations also let us answer the second question – evaluate the impact of different optimizations on the *netmap*'s performance.

Data copies, as shown in Section 5.6, are moderately expensive, but they do not prevent significant speedups (such as the 7.5 Mpps achieved forwarding packets on top of libpcap+netmap).

Per-packet system calls certainly play a major role, as witnessed by the difference between netsend and pktgen (albeit on different platforms), or by the low performance of the packet generator when using small batch sizes.

Finally, an interesting observation on the cost of the skbuf/mbuf-based API comes from the comparison of pktgen (taking about 250 ns/pkt) and the netmap-based packet generator, which only takes 20-30 ns per packet which are spent in programming the NIC. These two application essentially differ only on the way packet buffers are managed, because the amortized cost of system calls and memory copies is negligible in both cases.

5.8 Application porting

We conclude with a brief discussion of the issues encountered in adapting existing applications to *netmap*. Our libpcap emulation library is a drop-in replacement for the standard one, but other performance bottlenecks in the applications may prevent the exploitation of the faster I/O subsystem that we provide. This is exactly the case

we encountered with two applications, OpenvSwitch and Click (full details are described in [19]).

In the case of OpenvSwitch, the original code (with the userspace/libpcap forwarding module) had a very expensive event loop, and could only do less than 70 Kpps. Replacing the native libpcap with the netmap-based version gave almost no measurable improvement. After restructuring the event loop and splitting the system in two processes, the native performance went up to 780 Kpps, and the netmap based libpcap further raised the forwarding performance to almost 3 Mpps.

In the case of Click, the culprit was the C++ allocator, significantly more expensive than managing a private pool of fixed-size packet buffers. Replacing the memory allocator brought the forwarding performance from 1.3 Mpps to 3.95 Mpps when run over netmap, and from 0.40 to 0.495 Mpps when run on the standard libpcap. Click userspace is now actually faster than the in-kernel version, and the reason is the expensive device driver and sk_buff management overhead discussed in Section 2.3.

6 Conclusions and future work

We have presented *netmap*, a framework that gives userspace applications a very fast channel to exchange raw packets with the network adapter. *netmap* is not dependent on special hardware features, and its design makes very reasonable assumptions on the capabilities of the NICs. Our measurements show that *netmap* can give huge performance improvements to a wide range of applications using low/level packet I/O (packet capture and generation tools, software routers, firewalls). The existence of FreeBSD and Linux versions, the limited OS dependencies, and the availability of a libpcap emulation library makes us confident that netmap can become a useful and popular tool for the development of low level, high performance network software.

An interesting couple of open problems, and the subject of future work, are how the features of the netmap architecture can be exploited by the host transport/network stack, and how can they help in building efficient network support in virtualized platforms.

Further information on this work, including source code and updates on future developments, can be found on the project's page [17].

References

[1] The dag project. Tech. rep., University of Waikato, 2001.

[2] DERI, L. Improving passive packet capture: Beyond device polling. In *SANE 2004, Amsterdam*.

[3] DERI, L. ncap: Wire-speed packet capture and transmission. In *Workshop on End-to-End Monitoring Techniques and Services* (2005), IEEE, pp. 47–55.

[4] DOBRESCU, M., EGI, N., ARGYRAKI, K., CHUN, B., FALL, K., IANNACCONE, G., KNIES, A., MANESH, M., AND RATNASAMY, S. Routebricks: Exploiting parallelism to scale software routers. In *ACM SOSP* (2009), pp. 15–28.

[5] HAN, S., JANG, K., PARK, K., AND MOON, S. Packetshader: a gpu-accelerated software router. *ACM SIGCOMM Computer Communication Review 40*, 4 (2010), 195–206.

[6] HANDLEY, M., HODSON, O., AND KOHLER, E. Xorp: An open platform for network research. *ACM SIGCOMM Computer Communication Review 33*, 1 (2003), 53–57.

[7] HEYDE, A., AND STEWART, L. Using the Endace DAG 3.7 GF card with FreeBSD 7.0. *CAIA, Tech. Rep. 080507A, May 2008.[Online]. Available: http://caia. swin. edu. au/reports/080507A/CAIA-TR-080507A. pdf* (2008).

[8] INTEL. Intel data plane development kit. *http://edc.intel.com/Link.aspx?id=5378* (2012).

[9] JACOBSON, V., AND FELDERMAN, B. Speeding up networking. *Linux Conference Au, http://www.lemis.com/grog/Documentation/vj/lca06vj.pdf*.

[10] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. The click modular router. *ACM Transactions on Computer Systems (TOCS) 18*, 3 (2000), 263–297.

[11] KRASNYANSKY, M. Uio-ixgbe. *Qualcomm, https://opensource.qualcomm.com/wiki/UIO-IXGBE*.

[12] LOCKWOOD, J. W., MCKEOWN, N., WATSON, G., ET AL. Netfpga–an open platform for gigabit-rate network switching and routing. *IEEE Conf. on Microelectronics Systems Education* (2007).

[13] LWN.NET ARTICLE 192767. Reconsidering network channels. *http://lwn.net/Articles/192767/*.

[14] MCCANNE, S., AND JACOBSON, V. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX Winter Conference* (1993), USENIX Association.

[15] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review 38* (March 2008), 69–74.

[16] MOGUL, J., AND RAMAKRISHNAN, K. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems (TOCS) 15*, 3 (1997), 217–252.

[17] RIZZO, L. Netmap home page. *Università di Pisa, http://info.iet.unipi.it/∼luigi/netmap/*.

[18] RIZZO, L. Polling versus interrupts in network device drivers. *BSDConEurope 2001* (2001).

[19] RIZZO, L., CARBONE, M., AND CATALLI, G. Transparent acceleration of software packet forwarding using netmap. *INFOCOM'12, Orlando, FL, March 2012, http://info.iet.unipi.it/∼luigi/netmap/*.

[20] SCHUTZ, B., BRIGGS, E., AND ERTUGAY, O. New techniques to develop low-latency network apps. *http://channel9.msdn.com/Events/BUILD/BUILD2011/SAC-593T*.

[21] SOLARFLARE. Openonload. *http://www.openonload.org/* (2008).

[22] STEVENS, W., AND WRIGHT, G. *TCP/IP illustrated (vol. 2): the implementation*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

Toward Efficient Querying of Compressed Network Payloads

Teryl Taylor
UNC Chapel Hill
tptaylor@cs.unc.edu

Scott E. Coull
RedJack
scott.coull@redjack.com

Fabian Monrose
UNC Chapel Hill
fabian@cs.unc.edu

John McHugh
RedJack
john.mchugh@redjack.com

Abstract

Forensic analysts typically require access to application-layer information gathered over long periods of time to completely investigate network security incidents. Unfortunately, storing longitudinal network data is often at odds with maintaining detailed payload information due to the overhead associated with storing and querying such data. Thus, the analyst is left to choose between coarse information about long-term network activities or brief glimpses of detailed attack activity. In this paper, we take the first steps toward a storage framework for network payload information that provides a better balance between these two extremes. We take advantage of the redundancy found in network data to aggregate payload information into flexible and efficiently compressible data objects that are associated with network flows. To enable interactive querying, we introduce a hierarchical indexing structure for both the flow and payload information, which allows us to quickly prune irrelevant data and answer queries directly from the indexing information. Our empirical results on data collected from a campus network show that our approach can significantly reduce the volume of the stored data, while simultaneously preserving the ability to perform detailed queries with response times on the order of seconds.

1 Introduction

A complete incident response strategy for network attacks includes short-term detection and mitigation of the threat, as well as a broad forensic process. These forensic investigations attempt to discover the root cause of the attack, its impact on other resources, and how future attacks may be prevented. To perform these forensic tasks, however, a security analyst needs long-term, detailed information on the activities of monitored resources, which often includes the application-layer communications found within packet payloads, such as DNS queries and HTTP responses.

These requirements are perhaps best illustrated by the recent tide of attacks by so-called advanced persistent threats on major corporations and government contractors (eg., Google, RSA and Oakridge National Laboratory [30]). In each case, a single security breach (e.g., phishing or browser exploit) was used to gain a foothold within an otherwise secure network, which remained undetected for weeks while computing resources were accessed and proprietary information was exfiltrated. The only way to grasp the full impact of these types of attacks is to trace through each step and examine the associated communications information, including DNS names, HTTP redirection, and web page requests.

The obvious problem is that modern enterprise networks can easily produce terabytes of packet-level data each day, which makes efficient analysis of payload information difficult or impossible even in the best circumstances. Although several approaches have been developed to capture and store network flow (i.e., NetFlow) connection summaries (e.g., [6, 8, 12, 13, 16, 24]), these systems are inappropriate to the task at hand since they necessarily discard the application-layer information that is so important to the forensic analysis process. Furthermore, packet payload data introduces several unique challenges that are not easily addressed by traditional database systems. For one, there are thousands of application-layer protocols, each with their own unique data schemas[1]. Many of these protocol schemas are also extremely dynamic with various combinations of fields and use of complex data types, like arrays. Of course, there are also cases where the existence of proprietary or unknown application-layer protocols may prevent us from parsing the payload at all, and yet we still must be able to support analysis of such data.

Using a standard relational database solution in this scenario, be it column or row-oriented, would require thousands of structured tables with tens or hundreds of sparsely populated columns. Consequently, these tradi-

[1] As a case in point, Wireshark can parse over 10,000 protocols, fifty percent of which have more than 20 fields.

tional approaches introduce significant storage overhead, require complex join operations to relate payload fields to one another, and create bottlenecks when querying for complex payload data (e.g., arrays of values). While the availability of distributed computing frameworks [5, 11] help tackle some of these issues, the query times required for interactive analysis might only be realized when hundreds of machines are applied to the task – resources that may be unavailable to many analysts. Moreover, even when such resources are available, data organization remains a critical issue. Therefore, the question of how to enable efficient and interactive analysis of long-term payload data remains.

In this work, we address these challenges by extending existing network flow data storage frameworks with a set of summary payload objects that are highly compressible and easy to index. As with many of the network flow data stores, we aggregate network data into bidirectional flows and store the network-level flow data (e.g., IP addresses, ports, timestamps) within an indexed, column-oriented database structure. Our primary contribution, however, lies in how we store and index the payload information, and then attach that information to the column-oriented flow database. By using our proposed storage technique we are able to reduce the volume of data stored and still maintain pertinent payload information required by the analyst. While our ultimate goal is to create a data store for arbitrary payload content, we begin our exploration of the challenges of doing so by focusing on the storage and querying of packet payloads with well-defined header and content fields (e.g., DNS and HTTP). These protocols have many of the same issues described above, but represent a manageable step toward a more general approach to efficient payload querying.

To achieve our performance goals, we move away from the relational model of data storage and instead roll-up packet payloads into application-layer summary objects that are encoded within a *flexible, self-describing object serialization framework*. In this regard, our contributions are threefold. *(1)* We index the summary objects by treating them as documents with the application-layer field-value pairs acting as words that can be indexed and efficiently searched with bitmap indexes. This approach allows us to store and index heterogenous payload information with highly-variable protocol schemas — a task that is difficult, if not impossible, with previous network data storage systems. The bitmap indexes also make it possible to relate the payload objects to network flow records using only simple bit-wise operations. (*2*) We introduce a *hierarchical indexing* scheme that allows us to answer certain queries directly from in-memory indexes without ever having to access the data from disk, thereby enabling the type of lightweight iterative querying needed for forensics tasks. (*3*) We take advantage of the inherent redundancy found in many types of application-layer traffic to devise an effective *dictionary-based string compression* scheme for the payload summary objects, while standard block-level lossless compression mechanisms are utilized to take advantage of inter-field relationships in objects in order to minimize storage overhead and disk access times.

Given the privacy-related challenges in gaining access to network data containing complete payload information, we chose to evaluate our approach with two datasets containing DNS and truncated HTTP data that we collected at the University of North Carolina (UNC). The first dataset contains over 325 million DNS transactions collected over the course of five days. We use that dataset to compare our performance to that of the `SiLK` network flow toolkit [25] and Postgres SQL database, both of which are commonly used within the network analysis community to perform security and forensic analysis tasks [14, 23]. The second dataset is collected over 2.5 hours and contains over 11 million DNS and HTTP connections making up 400GB of packet trace data. We use that dataset to explore how well our approach handles heterogenous objects. The results of our experiments show that our approach reduces the volume of the DNS traffic that need be stored by over 38% and HTTP traffic by up to 97%, while still preserving the ability to perform detailed queries on the data in a matter of seconds. By comparison, using a relational database approach containing tables for the flow and payload schemas *increases* the data volume by over 400% and results in extremely slow response times.

2 Background and Related Work

One of the most common data storage approaches for network data is the so-called scan-and-filter paradigm. In this approach, raw network data is stored in flat files with standardized formats. To fulfill a given query, the analysis tool reads the entire data file from disk, scans through the contents, and applies one or more filtering criteria to select the pertinent records. The `SiLK` toolkit [25] and `TCPDump` are examples of this approach for network flow logs and packet traces, respectively. The ownside is that it requires the entire dataset to be read from disk, which is a relatively low-bandwidth process.

The simple scan-and-filter approach can be improved through the use of indexing methodologies to target disk access to only those records that meet the requirements of the user's query. As an example, the `TimeMachine` system [17] stores truncated packet data and uses hash-based indexing of packet headers to improve performance. However, `TimeMachine` is unable to index payload contents and each file found with the index must still be read from disk in its entirety even if only a single packet is requested.

One natural extension to the above is to divide packet data across machines and parallelize queries using distributed computing frameworks, such as `MapReduce` [11] or `Hadoop` [5]. Unfortunately, the computing resources necessary to take advantage of these frameworks are not always easily accessible to forensic analysts, and even organizations with significant computational resources (e.g., Google, Twitter) have acknowledged that simply parallelizing brute force data retrieval methods will not provide adequate performance [19]. In fact, Google developed a non-relational, column-oriented data store, called Dremel [18], specifically to address the issue of efficient data organization in distributed computing environments. Even so, Dremel does not support indexes and still resorts to scan-and-filter approaches to match attributes from different columnar files.

Beyond the simple scan-and-filter approach, traditional relational databases are also often used to store and query network data [9, 15]. A relational database allows data that conforms to the same static schema to be grouped together into tables, often with each row of data stored contiguously on disk (i.e., row-oriented). These databases provide a generic framework for storing a wide variety of data, and offer advanced indexing features to pinpoint exactly the records requested by the user. Although the relational databases offer these useful features, the row-oriented organization of the data is inefficient for network data queries that inspect only a small subset of discontiguous fields [6]. Additionally, the rigid schema structure of the database would result in hundreds of tables (one per application-layer protocol) with sparsely filled columns that require the use of expensive join operations to retrieve payload fields.

A more efficient approach for storing network data is to use a column-oriented database, where each column is stored independently as a sequence of contiguous values on disk. Column-oriented databases follow the same relational structure of rigid data tables, but enable efficient queries over multiple fields. For that reason, several column-based data stores have been developed specifically for network flow data [6, 12, 16, 24]. These systems create columns for each of the standard fields found within all flow records (e.g. IP, port, etc.), and then apply indexing methods to quickly answer multidimensional queries over several fields. When considering the variety and variability of payload data, however, the column-oriented approach suffers from some of the same shortcomings as row-oriented relational databases; namely, they require thousands of sparsely populated columns and table joins to store payload data for the most common application-layer protocols.

To our knowledge, only one other work has examined the problem of enabling forensic analysis of packet payload data. In this work, Ponec et al. [21] discuss how to capture short, overlapping windows of payload data and encode that data in memory-efficient data structures, similar to Bloom filters. The analyst then queries the data structure for a known byte sequence to determine if the sequence occurs within the data. While this solution can significantly reduce the storage requirements for payload data, it limits opportunities for exploratory data analysis since it can only be used to determine whether a previously-known byte sequence exists and cannot actually return the raw data (or any summaries thereof).

In designing our approach we purposely move away from the rigid relational database paradigm and instead draw inspiration from document-oriented databases (e.g., [1, 3]) that store self-describing document objects with flexible data schemas. By using this approach, we accommodate the strongly heterogenous nature of payload data and the variability of application-layer protocols. We combine these technologies with proven column-oriented database technologies, like bitmap indexes [13], to create a hybrid system that combines efficient storage of flow data in a column-oriented format with flexible storage of packet payload data.

3 Approach

The primary goal of our network data storage system is to enable fast queries over both network flow data and packet payload information. Our intention in this work is to develop an approach that takes advantage of the properties of network data to intelligently reduce workload and make payload analysis accessible to analyst with limited resources. As a result, we focus our evaluation on a single machine architecture to gain a better understanding of the key bottlenecks in the query process so that they may be mitigated. At the same time, we ensure our framework can naturally scale to take advantage of distributed computing resources by partitioning the data and developing a hierarchical system of indexes for both flow and payload information.

Before delving into the specifics of our approach, we first provide a basic overview of the storage and retrieval components. In the storage component, shown in Figure 1, incoming packets and their payloads are collected into bidirectional network flows in a manner similar to that proposed by Maier et al. [17]. After a period of inactivity, the flows are closed and packet payloads are aggregated. If the application-layer protocol contained within the aggregated payload is known to the storage system, it dissects the protocol into its constituent fields and their values, resulting in a set of *flow* and *payload* fields. When a sufficient number of closed flows have accumulated, the flows are indexed and written to one of many horizontal data partitions on the disk. The flow fields are written into a column-oriented store, while the associated payload attributes are serialized and compressed into flexi-

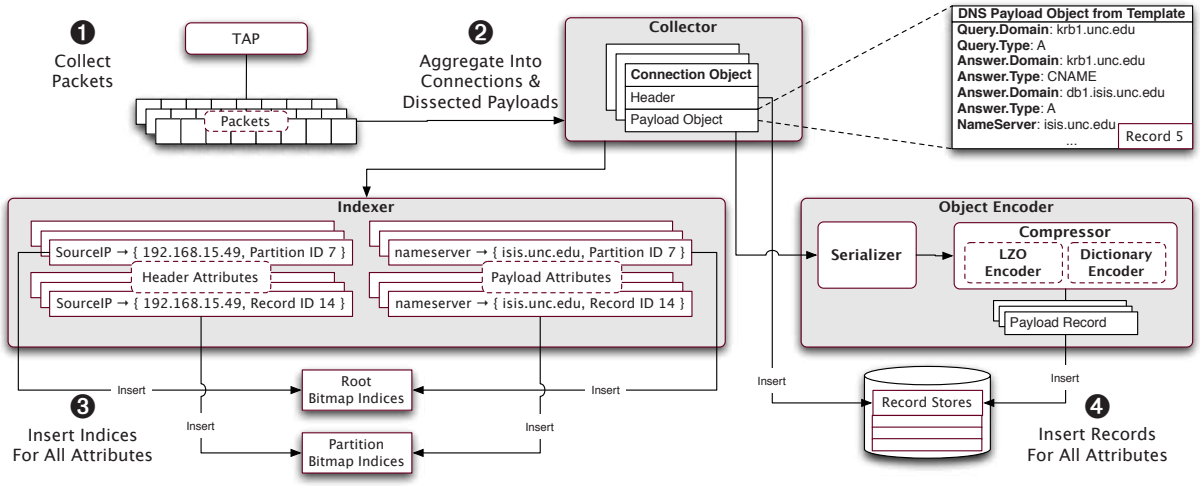


Figure 1: Storing payload data.

ble payload objects. Meanwhile, the flow and payload attributes are indexed in a hierarchical indexing system where the *root index* collects coarse information about the entire data store and the *partition indexes* collect more refined information about each of the horizontal partitions that the data is split into.

The analyst queries the data store using a SQL-like language with extensions that allow queries on payload data and queries that are answered directly from the hierarchical indexes. In Figure 2, for example, the analyst issues a query to find source IP addresses and DNS query types (e.g., MX records) for all traffic with destination port 53 and domain `www.facebook.com`. The query predicates (e.g., destination port and domain) are first compared to the in-memory root index to quickly return a list of data partitions that contain at least one record that matches those predicates. Next, the indexes for the matching partitions are examined to determine the exact locations of the records that match the given query predicate. Finally, the data store seeks to the given locations and retrieves the source IP from its column and the query type from the packet payload object.

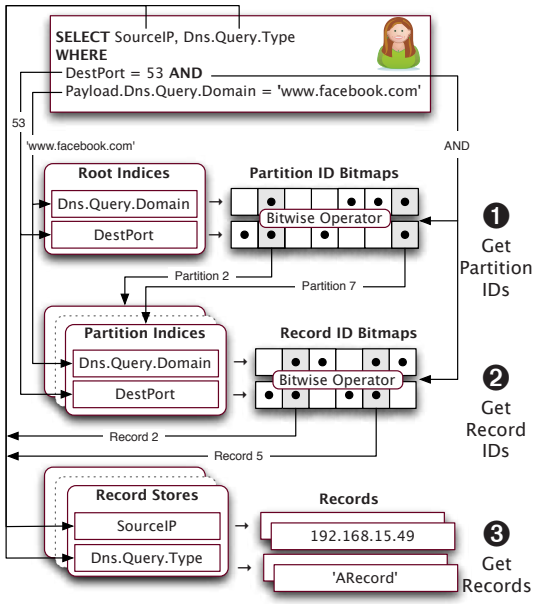


Figure 2: Processing a query.

3.1 Storage

Aggregation and Partitioning. Intelligently storing data on disk improves performance by simultaneously reducing the storage overhead and the amount of data that must be retrieved from high-latency disks. In order to reduce the data footprint, we aggregate all of the packets in a connection into network flows according to the five-tuple of source and destination IP address, source and destination ports, and protocol. As with previous approaches [6, 12, 13, 24], we utilize a column-oriented data store for the standard set of integer-based flow fields that occur in every network flow record[2], as it provides the best performance for the type of random-access queries made by forensic analysts [6, 23].

Our approach to storing payload information is to receive dissected payloads from one or more sensor platforms (e.g, Bro [28]) and extract the values from certain fields deemed important by the analyst. These field values are then stored in summary payload objects that

[2]Currently we support attributes of source and destination IP, source and destination ports, protocol, start time, duration, TCP flags, byte and packet count, but other attribute can be incorporated easily.

are instantiated from a set of application-layer protocol templates as illustrated by the DNS template example in Figure 3. Once the object is generated in memory, it is serialized to disk with a lightweight object serialization framework with self-describing data schemas [4]. The object serialization framework allows us to accommodate for the strongly heterogenous nature of payload data by individually defining the output schema for each payload object we serialize based on the extracted fields and values, rather than forcing the data to adhere to a static schema, as is the case in traditional relational databases. Moreover, if the object dissector encounters a payload for which it does not have an object template, we can still store the raw payload data in the same way by using an object with a single field (i.e., "raw payload") whose value is the byte sequence for the aggregated payload.

One unique feature of network data storage is that it does not require the updating capabilities normally associated with traditional databases. That is, once the data is written to disk, there is no need to update it. To take advantage of this property, we implement a horizontal partitioning scheme where groups of records are broken into independent partitions. In doing so, we can write all records in a partition at once, create indexes on that data, and never have to perform writes to that partition or index ever again. The use of horizontal partitioning also has the added benefit of providing natural boundaries where the data can be distributed among multiple servers or archived when space is limited.

```
struct dns_rr {
    std::string domain;
    proto::RecordType type;
    int32_t ttl;
    std::string response;
};

struct DNSMessage {
    std::string queryDomain;
    int32_t queryId;
    proto::RecordType recordType;
    array<proto::dns_rr> additional;
    array<proto::dns_rr> answer;
    array<proto::dns_rr> nameServer;
};
```

Figure 3: Example Object Definition.

Compression. Aside from aggregation of packet information into flow-based fields and payload objects, the nature of many application-layer protocols makes them particularly well-suited to the use of lossless compression mechanisms. In particular, many application-layer protocols, such as DNS and HTTP, exhibit high levels of string redundancy [10]. To take advantage of this redundancy, we apply two compression mechanisms to our payload data to ensure we need only read a minimum of information from disk.

The first compression scheme employs a dictionary-based encoding of all strings in the payload objects. In a dictionary-based encoding scheme, long byte strings are replaced with much smaller integer values that compress the space of strings accordingly. Undoubtedly, we are not the first to take advantage of this observation. Binnig et al. [7], for example, showed that string compression can be effective even for data with extremely high cardinality. However, their approach cannot be readily applied in our setting as it is not well suited for streaming data sets, like network traffic. Fortunately, we found that a rather straightforward hash-based approach that maps strings to integer values as the packets are dissected works extremely well in practice.

Our second compression mechanism uses standard Lempel-Ziv-Oberhumer (LZO) encoding compression [32] to compress the serialized payload objects after they have been dictionary-encoded. LZO is notable for its decompression speed, which is ideal for network data that will be written once and read multiple times. It is also particularly efficient at compressing highly-redundant but variable-length patterns.

As we show later, this integration of compression techniques dramatically reduces response times through reduced data footprint, and helps offset the storage overhead introduced by indexing methods that help us quickly retrieve data.

3.2 Indexing and Retrieval

Avoid Scan-and-Filter. Scan-and-filter approaches to data retrieval are slow because of the need to read the entire dataset to find records of interest. The standard solution to this problem is to apply indexing to the data to quickly determine the location of the data that satisfies the query. Construction of these indexes, however, requires careful planning in our setting. For one, network data is multi-dimensional and an analyst will often not know exactly what she is looking for until she has found it [6, 23]. Therefore, we must support indexes across all of the stored flow and payload fields, as well as combinations of those fields. Furthermore, it is also important that these indexes can be built quickly and effectively support large data sets.

In this work, we use two separate indexing mechanisms; one for flow fields and another for payload fields. For flow fields, we use bitmap-based indexes [6, 12, 13, 24]. Simply put, a bitmap index is a pair containing a field value and a bit vector where a bit at position i indicates the presence or absence of that value in record i. The bitmap indexes benefit from being highly-compressible [31], enabling combination of in-

dexes through fast bit-wise operations (i.e., AND and OR), and allowing real-time generation as data is being stored [13]. Each flow field stored has its own bitmap index. We refer the interested reader to Deri et al. [12] for an in-depth discussion on bitmap indexes.

In contrast, packet payload data is far more difficult to index because it is composed of an arbitrary number of attributes that have high cardinality and varying length. Therefore, we cannot take the same approach used in indexing flow data. However, we can model our summary packet payload objects as documents by considering the field-value pairs (e.g., '`query domain = www.facebook.com`') as words to be indexed. Specifically, we create an inverted-term index that stores the field-value pairs across all payload objects in lexicographical order, grouped by field name. Each field-value pair is associated with a bitmap indicating the records where the pair is found, as shown in Figure 4. In this way, we can now support the ability to search for specific criteria within the payload, including wildcards (e.g., '`Http.UserAgent:Mozilla/5.0*`'). Additionally, the payload bitmap index can be combined with those of the network flow columns to tie both data stores together.

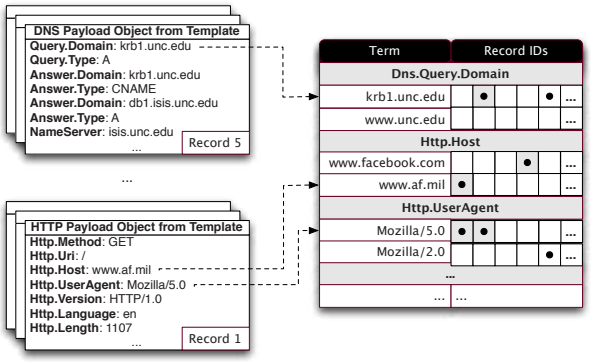


Figure 4: Indexing packet payload content.

Abstraction via Indirection. Although it is possible to use a single, monolithic index for each of the flow and payload fields in the data store, they are likely to become unmanageably large after a few million records have been added. Once the cardinality of those indexes becomes sufficiently large, even reading the index from disk would take a non-trivial amount of time and would likely be difficult to fit into memory. Another approach would be to have indexes associated with each of the horizontal partitions of the data. In that case, one would incur significant disk access penalties by reading the indexes for every partition even if it contains no records of interest. To address these issues, we designed a hierarchical indexing approach, similar to that of Sinha and Winslett [26], to efficiently exclude partitions that cannot satisfy a query. Our framework uses a root index to pick candidate partitions, and then processes the partition-level indexes to resolve queries to specific records.

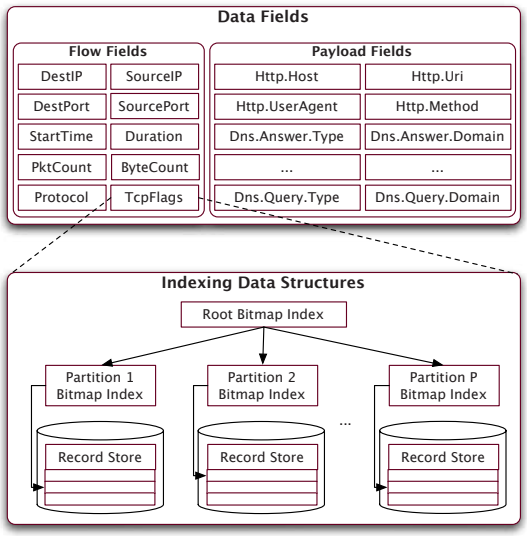


Figure 5: Structure of the datastore

Each of the network flow columns, plus the payload store, has its own root index for locating partitions that satisfy queries, as well as partition-level indexes for locating matching records. Both the root and partition-level indexes are organized as described above, with each value associated with a bitmap indicating the partitions and records it occurs in, respectively. The key difference between the root and partition index is that the root index is constantly updated as records are added to the data store, whereas partition-level indexes are written all at once when the partition is written to disk. Therefore, we make use of a B-tree data structure at the root index for each flow field, which allows us to efficiently insert and update the bitmaps associated with the field values. Meanwhile, the root index for payload objects is structured as a document index similar to those found in the partitions, except that the bitmaps for each of the field-value pairs point to the appropriate partitions.

The root and partition-level indexes are ideal for finding attributes that appear rarely in the data. Therefore, we may use them to answer counting queries using only the in-memory indexes. The advantage of these types of index-only queries is that they are relatively fast compared to standard queries because they do not have to access data from high-latency disks. One example of such a query might be: '`SELECT Count(*) WHERE Protocol=6 AND DestPort=80`'.

The ability to support index-only queries is important, if for no other reason than it improves the interactive feedback loop that is common in forensic analy-

sis [6, 23]. There, the typical modus operandi is to start with a single query that may return far too many records, and then repeatedly refine the search criteria until an acceptable number of matches are brought forth. Note also that it is possible to extend the current two-level index hierarchy to an arbitrary number of levels, which facilitates indexing of partitions that are hierarchically distributed within a computing cluster, thereby enabling multiple levels of resolution in the index-only queries.

4 Evaluation

In this section, we present an empirical evaluation of our system using two separate network traces. First, we collected approximately 122 GBs of DNS traffic between the UNC DNS servers and external servers during five days in March of 2011. We also collected over 400 GBs of DNS and truncated HTTP traffic over a few hours during a week day in January 2012. The specifics of both datasets are shown in Table 1.

Trace 1: DNS traffic	
Length of Trace	5 days
Average DNS queries per day	66.5 M
Average no. of clients per day	272,945
Original raw trace	122 GB
LZO compressed raw trace	55 GB
Data store (uncompressed)	155 GB
Data store (dict. compression)	83 GB
Data store (dict. + LZO)	75 GB
Trace 2: DNS + HTTP traffic	
Length of Trace	2.5 hrs
Number of Connections	11.1 M
Original raw trace	400 GB
LZO compressed raw trace	358 GB
Data store	12 GB
Number of payload fields	1700+
Number of distinct payload field values	11 M+

Table 1: Data Summary

Our framework was built as a shared-library and was designed to receive events from front-end data collection and sensor products. For the purposes of our evaluation, we integrated our framework with the Bro Intrusion Detection System and collected both DNS and HTTP events. The packets from these events were aggregated offline into flow records with associated payload objects, as described in Section 3. The size of each of our horizontal partitions was set to one million records ($k = 1M$). We chose this value based on empirical results that showed that this choice provided a reasonable balance between large index file sizes and the overhead induced by opening large numbers of files.

All experiments discussed in this section were performed on an Ubuntu Linux server with dual Intel Xeon 2.27 GHz processors, 12 GBs RAM, and a single 2TB 7200 RPM local SATA drive. We chose this configuration because it resembles platforms typically used by researchers and practitioners for network forensic investigation. The results of our experiments were averaged over five runs of each query. Memory and disk caches were cleared between each query.

4.1 Query Types

To better assess the benefits of our approach in quickly retrieving network data, we performed a series of queries that span three categories typically seen in forensic investigations. These queries are similar to those found in previous work focusing on the storage and retrieval of network data [6, 17, 20, 24], and represents the types of queries used by security analysts [23]:

- *Heavy Hitters*: Returns the majority of records in the datastore. Such queries are typically used to gather global statistics about the dataset. This class of queries serves as a good stress test for our approach, and also serves as a point of comparison for sequential scanning techniques. Heavy Hitter queries are also used as a baseline for showing how indexes can affect query times. An example of such a query might be: '`SELECT SourceIP WHERE Protocol = 17 OR Protocol=6`'.

- *Partition Intensive*: Returns records from each partition, but not the majority of the records from those partitions. Analysts might use these types of queries in the early stages of their investigations (e.g., when looking for a specific IP address responsible for a significant amount of traffic or for activity on a common port). Partition Intensive queries are used to show the speedup achieved because of our indexing structures. An example of such a query might be: '`SELECT Dns.Query.Type, Dns.Query.Id WHERE Payload.Dns.Query.Domain = www.facebook.com`'.

- *Needle in a Haystack*: Returns a few records from the datastore. These types of queries might arise in cases where an analyst is searching for a rare event (e.g., traffic to a rogue external host on certain ports). This class of queries demonstrate the effectiveness of hierarchical indexing, as well as the overhead involved with the querying system. An example of such a query might be: '`SELECT SourceIP, SourcePort WHERE Payload.Dns.Query.Domain = www.dangerous.com`'.

4.2 Results

Our first set of experiments were performed on the 122 GB DNS traffic trace. This dataset is interesting because it has a large volume of connections (over 325 million). Since the DNS payload objects store almost all attributes from the DNS packets, the data stored within our system is extremely dense (i.e., large number of payload attributes per flow record), and thus serves as a good test of data retrieval capabilities.

To gain a deeper understanding of the parameters that affect overall query performance, we first vary the number of attributes ($n \in \{1,2,4\}$) returned in the `SELECT` clause while keeping the number of attributes specified in the `WHERE` clause constant. Furthermore, we measure the differences between flow and payload attributes in the queries. Four queries were issued for each of the categories listed in Section 4.1. These queries return flow and payload attributes using the available indexes.

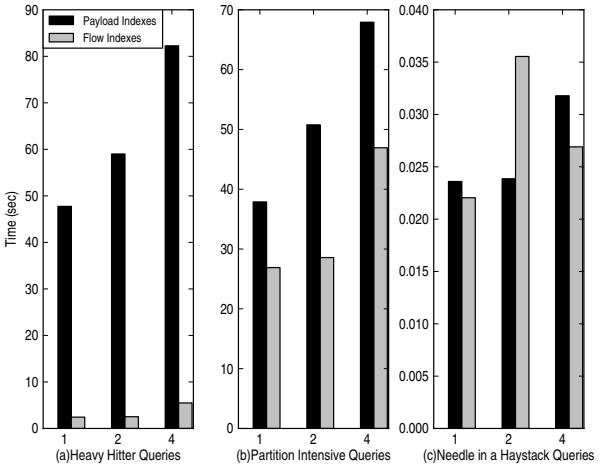


Figure 6: Response times for returning flow attributes filtered using flow indexes (grey) and the payload index (black).

Returning Flow Attributes. Figure 6 (grey bars) shows the performance when returning 1, 2, or 4 flow-based attributes filtered by a flow index. Response times are fast because queries operate on small column files and indexes; therefore, there is no parsing overhead and disk I/O times are reduced. On average, Heavy Hitter query times using flow indexes (6(a)) are lower than Partition Intensive queries (6(b)). While this may seem odd, the results can be explained by the fact that there is significant overhead associated with reading the many attribute indexes used in the `WHERE` clause of the Partition Intensive query, whereas the Heavy Hitter query used only a single, low-cardinality index.

Figure 6 (black bars) shows the query performance when returning 1, 2, or 4 flow-based attributes filtered by the payload index. These types of queries are slower because payload indexes are much larger in size than those for flows because of their cardinality and the variability in length of the indexed values. Even though the queries are slower, notice that even a Heavy Hitter query that returns well over 200M records ($n = 4$ in Figure 6(a)) still completes in roughly one minute. In addition, because our hierarchical indexes efficiently prune irrelevant partitions, Needle in the Haystack queries are extremely fast in all cases — each with sub-second response times.

Returning Payload Attributes. Next, we investigate the query performance when returning payload-based attributes instead of flow-based attributes in the query. Payload-based queries are slower than flow-based queries because payload object files are larger, and such objects have parsing overhead. In order to improve query performance, we applied various compression techniques and compared the resulting query times. Specifically, we compared uncompressed, dictionary compressed, and dictionary+LZO compressed payload versions of our data store. Varying n in our experiments had little impact since the majority of the performance bottleneck can be attributed to loading and parsing the payload objects. Therefore, we only show results for $n = 2$ since the results for $n = 1,4$ are similar.

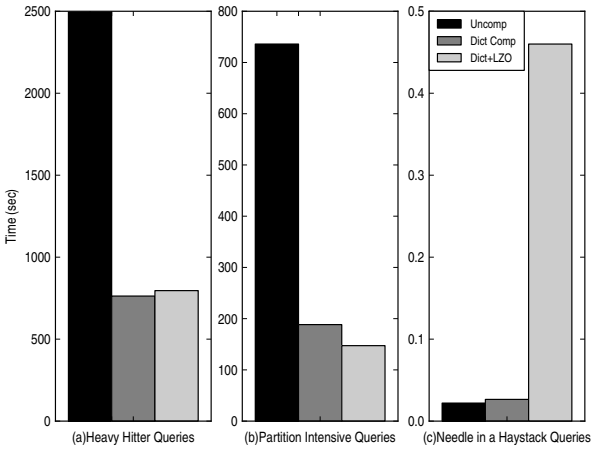


Figure 7: Return payload attributes filtered using a flow-based index (payload queries $n = 2$).

Figures 7 and 8 shows the performance on payload queries for $n = 2$ in experiment. For Heavy Hitter queries on flow indexes (Figure 7), we achieve more than 3 times speedup with dictionary-based and LZO compression enabled, and a 5-fold improvement for Partition Intensive queries. This improvement is a direct result of the reduction in the size of payload files as stored on disk. Needle in a haystack queries, on the other hand, retrieve little data from disk so there is little overhead when operating on uncompressed stores. In fact, the opposite is true: dictionary+LZO compressed queries are slower because the

storage manager must decompress the entire file before reading payload records from disk.

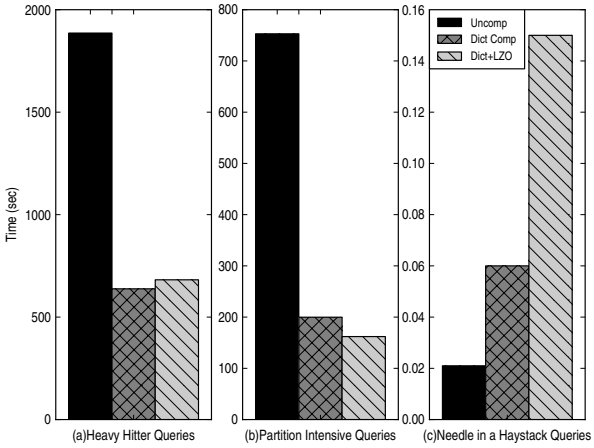


Figure 8: Return payload attributes filtered using a payload-based index (payload queries $n = 2$).

Figure 8 shows the results for payload-based queries using payload indexes. The results suggests that the overhead of processing payload indexes has a small impact compared to the overhead of reading and parsing the payload objects. Heavy hitter, compressed payload queries are nearly 2.8 times as fast as uncompressed payloads, while Partition Intensive queries are over 4.6 times faster.

Overhead Analysis. Next, we consider the increase in size of our data store over time and the component-wise overhead when performing payload queries. Figure 9 shows the growth of various framework components (using dictionary+LZO compression) over the duration of data collection for the DNS dataset. Note that due to space constraints the results are depicted on a log-linear plot. The graph shows that components grow at a very slow linear scale as new data is added. Some components, like the root flow indexes and the dictionary used for string compression, experience incredibly slow growth because of the reuse of field values that naturally occurs in network protocols.

Table 2 presents the average processing time spent in various components for a set of payload-based Heavy Hitter queries using payload indexes. The results show that the majority of time is spent in decoding objects and performing memory management tasks related to creating the result sets to be returned to the client — both of which are areas where optimizations are required to further improve query performance.

Performance Comparison. For comparison purposes, we also examined the performance of traditional relational database systems (e.g., [9, 15]) and toolkits for network forensics [14]. Specifically, we use Postgres version 9.0 and `SiLK` version 2.4.5. In the case of Postgres, we created four tables: a table for flow attributes, and three tables to hold DNS Answer, Name Server, and Additional resource records, respectively (see Figure 3). The tables are linked to support joins, and indexes were generated on all fields. The resulting data store was almost 5 times as large as the original dataset, and took 8 days to generate. We used a custom application to efficiently query the Postgres database. For the `SiLK` experiments, we generated the data store using `rwflowpack`, and all records were partitioned hourly following the typical usage of `SiLK` [25]. The resulting datastore was 7.1GBs. For our queries, the accompanying `rwfilter` utility was used[3].

Component	%
Payload Object Decoding	37.0%
Memory Allocation for Result Set	30.0%
Payload I/O	20.0%
LZO Decompression	6.4%
Miscellaneous	3.4%
Processing Payload Indices	3.2%

Table 2: Processing breakdown of a Heavy Hitter payload query using the payload index.

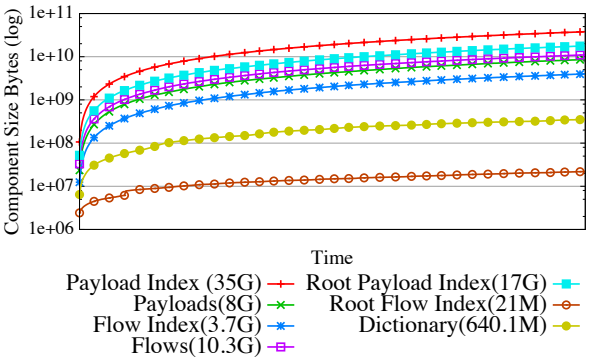


Figure 9: Growth of the components over time for Trace 1.

The results for a series of queries following the categories given in Section 4.1 are provided in Table 3. Notice that the relational database approach consistently performs the worst in all but the Needle in a Haystack queries on flow-based attributes. While still slower than our approach, the use of indexes enable it to avoid scanning all records as is the case with `SiLK`. `SiLK`'s performance remains constant across all the queries because it uses a sequential scan-and-filter approach whose performance is linear to the size of the data.

Our efforts to compare the performance of payload-based queries was also quite telling. While we had originally hoped to explore Heavy Hitter queries that involved joins between the flow table and the DNS response ta-

[3]For fairness, all output was directed to /dev/null to minimize overhead of console output.

Flow-based Queries	Heavy Hitters	Partition Intensive	Needle Haystack
Postgres	18.1m	9.5m	2.0s
SiLK	1.8m	1.8m	1.8m
Our approach	2.5s	30.5s	0.04s
Simple queries, no joins required			
Payload-based Queries	**Heavy Hitters**	**Partition Intensive**	**Needle Haystack**
Postgres	7.6m	9.7m	1.6s
Our approach	9.7m	3.3m	0.1s
Complex queries, joins required			
Payload-based Queries	**Heavy Hitters**	**Partition Intensive**	**Needle Haystack**
Postgres	> 2h	> 2h	3s
Our approach	30m	2.5m	0.1s

Table 3: Comparison to other approaches.

bles in the Postgres database, the response time was so slow that we terminated most of the queries after two hours (shown in Table 3 as $> 2h$). For a more simplified evaluation, we manually altered the flow table to simply include certain DNS-related fields directly (namely, domain and record type) and then issued queries directly on this altered table to avoid the costly join operations. In this case, the Heavy Hitter payload queries performed similarly in both Postgres and our data store. However, for multi-dimensional Partition Intensive and Needle in a Haystack queries, we again outperform Postgres, returning results in sub-second time in certain cases.

Storing Multiple Object Types. Having established the performance results of the proposed network data store, we now turn our attention to investigating the impact of heterogeneous objects on storage and query performance. In the experiments that follow, we use the DNS+HTTP dataset that contains over 400 GBs of DNS and truncated HTTP traffic. Due to privacy reasons, we were limited to storing only 500 bytes of each HTTP payload and were only allowed to collect that data for a short time period. To explore the impact of heterogeneous data, we use two summary payload objects in addition to the standard network flow store: the original DNS object (as in Figure 3) and an HTTP object which stored the method, URI, and all other available request and response headers as allowed by the truncated HTTP packets. All fields were then indexed and compressed.

Unlike the DNS dataset examined earlier in this section, this dataset contains only 11.1 million connections and a large portion of the traffic contents (i.e., actual web content) are excluded from the payload storage, making the dataset far less dense in terms of stored information. As a result, the 400GB packet trace is converted into a 12GB data store, including indexes and compressed data files. We tested the query performance of the more diverse data store using a select set of queries that mirror the three query classes used earlier. Our Heavy Hitter query returned all 11.1 million records in the data store in 52 seconds on average. The Partition Intensive query returned 6,000 records in 7.6 second on average. Finally, the Needle in a Haystack query returned one HTTP and one DNS record in under 0.4 seconds.

We believe this extended evaluation aptly demonstrates the flexibility and querying power of our approach. In particular, it shows we can store and index arbitrary object schemas, and provide high performance, robust queries across payloads. Furthermore, we are able to customize our summary payload objects and utilize compression techniques to considerably reduce the amount of data stored, while still storing important fields that are useful for forensic analyses.

5 Case Study

As alluded to earlier, post-mortem intrusion analysis has become an important problem for enterprise networks. Indeed, the popular press abounds with reports documenting rampant and unrelenting attacks that have led to major security breaches in the past. Unfortunately, many of these attacks go on for weeks, if not months, before being discovered. The problem, of course, is that the deluge of data traversing our networks, coupled with the lack of mature network forensic platforms, make it difficult to uncover these attacks in a timely manner. In order to further showcase the utility of our framework, we now describe how it was used in practice to identify UNC hosts within the DNS trace that contacted blacklisted domains or IP addresses. To aide with this analysis, we also obtained blacklists from a few sources, including a list of several thousand active malicious domains discovered by the Notos system of Antonakakis et al. [2].

First, for each entry in the blacklist, the root payload index was queried to assess whether these blacklisted domains appeared in the trace. Since the root index is sorted lexicographically by field value and there is no need to touch the partitions themselves, these queries return within milliseconds. The result is a bitmap indicating which partitions the domain appears in. Using these queries, we quickly pruned the list to 287 domains.

Next, we investigate how many records were related to DNS requests for these blacklisted domains by issuing a series of index-only queries (e.g., '`SELECT Count(*) WHERE DNS.QueryDomain = www.blacklisted.com`') to count the number of matching records. That analysis revealed that over 37,000 such requests were made within a one week period. Digging deeper, we were able to quickly pinpoint which internal IPs contacted the blacklisted domains (e.g., '`SELECT SourceIP, Time WHERE DNS.QueryDomain = www.blacklisted.com`'). To our surprise, we found at least one blacklisted request in *every* minute of the dataset. More interestingly, roughly

33% of those requests came from a single host attempting to connect to a particular well-known malicious domain name. Correlation with external logs showed that the machine in question was indeed compromised.

Encouraged by the responsiveness of the data store, we turned our attention to looking for additional evidence of compromises within the data. In this case, we issued wildcard queries for domains found among a list of several hundred domains extracted from forensic analysis of malicious PDF documents performed by Snow et al. [27]. Many of these domains represent malicious sites that use domain generation algorithms (DGAs)[4]. To search for the presence of such domains, we simply issued wildcard queries on payload fields. For instance, to find domain names from the `cz.cc` subdomain, which is known to serve malicious content [22, 29], we issued a query of the form: '`SELECT SourceIP, Time, DNS.QueryDomain WHERE DNS.QueryDomain = *.cz.cc`' and discovered 1,277 matches within the trace. While we strongly suspect that many of the connections identified represent traffic from compromised internal hosts, we are unable to confirm that without additional network traces. Nevertheless, all of the aforementioned analyses were conducted in less than fifteen minutes, and yielded valuable insights for the UNC network operators. Without question, the framework was particularly helpful in supporting interactive querying of network data.

6 Summary

Packet payload data contains some of the most valuable information for security analysts, and yet it remains one of the most difficult types of data to efficiently store and query because of its heterogeneity and volume. In this paper, we proposed a first step toward a fast and flexible data store for packet payloads that focuses on well-defined application-layer protocols, with a particular emphasis on DNS and HTTP data. To achieve these goals, we applied a combination of column-oriented data stores, flexible payload serialization, and efficient document-style indexing methods. Our evaluation showed that our approach for storage of payload content was faster and more flexible than existing solutions for offline analysis of network data. Finally, we underscored the utility of our data store by performing an investigation of real-world malware infection events on a campus network.

Overall, our evaluation brought to light several important insights into the problem of large-scale storage and querying of network payload data. The performance of our data store in returning flow attributes, for instance, serves as independent confirmation of the benefits of aggregating packet-level data and using column-oriented approaches to store data with well-defined schemas. Likewise, our results illustrated the power of hierarchical indexing and fixed-size horizontal partitions in both minimizing high-latency disk accesses and enabling the fast index-only queries that are key to interactive data analysis. It is also clear that document-based indexing and flexible object serialization are promising technologies for storing network payloads with highly dynamic data schemas and complex data types. This is particularly true when considering network data with high levels of redundancy, where we can use dictionary-based compression to limit storage overhead and the cardinality of payload indexes. Unfortunately, the document-oriented approach is not without its own pitfalls, since our evaluation also indicated that there are non-trivial amounts of overhead associated with deserializing the payload objects. Moving forward, we hope to build off of the insights from our offline data storage framework to incorporate real-time storage capabilities and to extend the system to distributed computing environment.

[4] See, for example, "*How Criminals Defend Their Rogue Networks*" at `http://www.abuse.ch/?tag=dga`

Acknowledgments

We are especially grateful to Michael Bailey, Douglas Creager, and Srinivas Krishnan for fruitful discussions regarding this work. Thanks to Hank Levy, Kevin Snow and the anonymous reviewers for their insightful comments and suggestions for improving an earlier draft of this paper. We also thank Murray Anderegg and Bil Hayes for their help in setting up the networking infrastructure that supported some of our data collection efforts. This work is supported in part by the National Science Foundation under award 1127361 and the Natural Sciences and Engineering Research Council of Canada (NSERC) Postgraduate Scholarship Program.

References

[1] 10gen Inc. MongoDB. Available at `http://www.mongodb.org/`.

[2] M. Antonakakis, R. Perdisci, D. Dagon, W. Lee, and N. Feamster. Building a Dynamic Reputation System for DNS. In *USENIX Security Symposium*, 2010.

[3] Apache Software Foundation. Apache CouchDB. See `http://couchdb.apache.org/`, 2008.

[4] Apache Software Foundation. Apache Avro. `http://avro.apache.org/`, 2011.

[5] Apache Software Foundation. Apache Hadoop. See `http://hadoop.apache.org/`, 2011.

[6] E. Bethel, S. Campbell, E. Dart, K. Stockinger, and K. Wu. Accelerating Network Traffic Analytics Using Query-Driven Visualization. In *IEEE Symposium on Visual Analytics Science And Technology*, pages 115–122, 2006.

[7] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD International Conference on Management of Data*, pages 283–296, 2009.

[8] E. Cooke, A. Myrick, D. Rusek, and F. Jahanian. Resource-aware multi-format network security data storage. In *SIGCOMM Workshop on Large-scale Attack Defense*, pages 177–184, 2006.

[9] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *ACM SIGMOD International Conference on Management of Data*, pages 647–651, 2003.

[10] C. Cunha, A. Bestavros, and M. Crovella. Characteristics of WWW Client-based Traces. Technical report, Boston University, 1995.

[11] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, 2008.

[12] L. Deri, V. Lorenzetti, and S. Mortimer. Collection and exploration of large data monitoring sets using bitmap databases. In *TMA*, pages 73–86, 2010.

[13] F. Fusco, M. Vlachos, and M. Stoecklin. Real-time creation of bitmap indexes on streaming network data. *The VLDB Journal*, pages 1–21, 2011.

[14] C. Gates, M. Collins, M. Duggan, A. Kompanek, and M. Thomas. More netflow tools: For performance and security. In *18th Conference on Systems Administration*, pages 121–132, 2004.

[15] R. Geambasu, T. Bragin, J. Jung, and M. Balazinska. On-demand view materialization and indexing for network forensic analysis. In *USENIX International Workshop on Networking Meets Databases*, pages 4:1–4:7, 2007.

[16] P. Giura and N. Memon. NetStore: An Efficient Storage Infrastructure for Network Forensics and Monitoring. In *Intl. Conf. on Recent Advances in Intrusion Detection*, pages 277–296, 2010.

[17] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider. Enriching network security analysis with time travel. In *ACM SIGCOMM Conference on Data Communication*, pages 183–194, 2008.

[18] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. In *Intl. Conf. on Very Large Databases*, 2010.

[19] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD international conference on Management of data*, pages 165–178, 2009.

[20] T. Plagemann, V. Goebel, A. Bergamini, G. Tolu, G. Urvoy-keller, and E. W. Biersack. Using data stream management systems for traffic analysis - a case study. In *Passive and Active Measurements*, pages 215–226, 2004.

[21] M. Ponec, P. Giura, J. Wein, and H. Brönnimann. New Payload Attribution Methods for Network Forensic Investigations. *ACM Transactions on Information Systems Security*, 13:1–32, 2010.

[22] N. Provos. Top 10 malware sites. Available at `http://googleonlinesecurity.blogspot.com/2009/06/top-10-malware-sites.html`, 2009.

[23] F. Reiss, K. Stockinger, K. Wu, A. Shoshani, and J. M. Hellerstein. Efficient analysis of live and historical streaming data and its application to cybersecurity. LBNL Technical Report 61080, 2006.

[24] F. Reiss, K. Stockinger, K. Wu, A. Shoshani, and J. M. Hellerstein. Enabling Real-Time Querying of Live and Historical Stream Data. In *International Conference on Scientific and Statistical Database Management*, page 28, 2007.

[25] T. Shimeall, S. Faber, M. DeShon, and A. Kompanek. *Analysts' Handbook: Using SiLK for Network Traffic Analysis*. CERT Network Situational Awareness Group, 2010.

[26] R. R. Sinha and M. Winslett. Multi-resolution Bitmap Indexes for Scientific Data. *ACM Transactions on Database Systems*, 32, August 2007.

[27] K. Z. Snow, S. Krishnan, F. Monrose, and N. Provos. Shellos: enabling fast detection and forensic analysis of code injection attacks. In *20th USENIX conference on Security*, pages 9–9, 2011.

[28] R. Sommer and V. Paxson. Enhancing Byte-level Network Intrusion Detection Signatures with Context. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 262–271, 2003.

[29] Sucuri Research. Malware from .cz.cc domains. Available at `http://sucuri.net/malware-from-cz-cc-domains.html`, 2011.

[30] J. Vijayan. Oak Ridge National Lab shuts down Internet, email after cyberattack. Available at `http://www.computerworld.com/`, 2011.

[31] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31:1–38, 2006.

[32] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. on Information Theory*, 24(5):530 – 536, 1978.

Body armor for binaries: preventing buffer overflows without recompilation

Asia Slowinska
Vrije Universiteit Amsterdam

Traian Stancescu
Google, Inc.

Herbert Bos
Vrije Universiteit Amsterdam

Abstract

BinArmor is a novel technique to protect existing C binaries from memory corruption attacks on both control data and non-control data. Without access to source code, non-control data attacks cannot be detected with current techniques. Our approach hardens binaries against both kinds of overflow, without requiring the programs' source or symbol tables. We show that *BinArmor* is able to stop real attacks—including the recent non-control data attack on Exim. Moreover, we did not incur a single false positive in practice. On the downside, the current overhead of *BinArmor* is high—although no worse than competing technologies like taint analysis that do not catch attacks on non-control data. Specifically, we measured an overhead of 70% for `gzip`, 16%-180% for `lighttpd`, and 190% for the `nbench` suite.

1 Introduction

Despite modern security mechanisms like stack protection [16], ASLR [7], and PaX/DEP/W⊕X [33], buffer overflows rank third in the CWE SANS top 25 most dangerous software errors [17]. The reason is that attackers adapt their techniques to circumvent our defenses.

Non-control data attacks, such as the well-known attacks on *exim* mail servers (Section 2), are perhaps most worrying [12, 30]. Attacks on non-control data are hard to stop, because they do not divert the control flow, do not execute code injected by the attacker, and often exhibit program behaviors (e.g., in terms of system call patterns) that may well be legitimate. Worse, for binaries, we do not have the means to detect them *at all*.

Current defenses against non-control data attacks all require access to the source code [20, 3, 4]. In contrast, security measures at the binary level can stop various control-flow diversions [15, 2, 19], but offer no protection against corruption of non-control data.

Even for more traditional control-flow diverting attacks, current binary instrumentation systems detect only the *manifestations* of attacks, rather than the attacks themselves. For instance, they detect a control flow diversion that *eventually* results from the buffer overflow, but not the actual overflow itself, which may have occurred thousands of cycles before. The lag between time-of-attack and time-of-manifestation makes it harder to analyze the attack and find the root cause [27].

In this paper, we describe *BinArmor*, a tool to bolt a layer of protection on C binaries that stops state-of-the-art buffer overflows immediately (as soon as they occur).

High level overview Rather than patching systems after a vulnerability is found, *BinArmor* is proactive and stops buffer (array) overflows in binary software, before we even know it is vulnerable. Whenever it detects an attack, it will raise an alarm and abort the execution. Thus, like most protection schemes, we assume that the system can tolerate rare crashes. Finally, *BinArmor* operates in one of two modes. In *BA-fields mode*, we protect individual fields inside structures. In *BA-objects mode*, we protect at the coarser granularity of full objects.

BinArmor relies on limited information about the program's data structures—specifically the buffers that it should protect from overflowing. If the program's symbol tables are available, *BinArmor* is able to protect the binary against buffer overflows with great precision. Moreover, in BA-objects mode no false positives are possible in this case. While we cannot guarantee this in BA-fields mode, we did not encounter any false positives in practice, and as we will discuss later, they are unlikely.

However, while researchers in security projects frequently assume the availability of symbol tables [19], in practice, software vendors often strip their code of all debug symbols. In that case, we show that we can use automated reverse engineering techniques to extract symbols from stripped binaries, and that this is enough to protect real-world applications against real world-attacks. To our knowledge, we are the first to use data structure

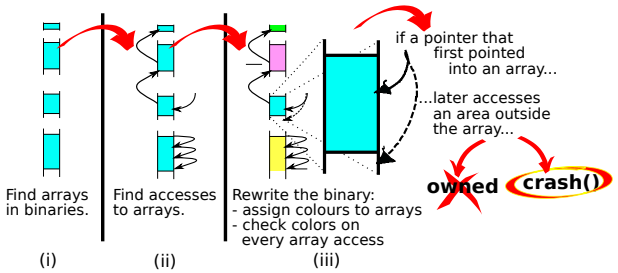


Fig. 1: *BinArmor* overview.

recovery to prevent memory corruption. We believe the approach is promising and may also benefit other systems, like XFI [19] and memory debuggers [24].

BinArmor hardens C binaries in three steps (Fig. 1):

(i) *Data structure discovery:* dynamically extract the data structures (buffers) that need protection.

(ii) *Array access discovery:* dynamically find potentially unsafe pointer accesses to these buffers.

(iii) *Rewrite:* statically rewrite the binary to ensure that a pointer accessing a buffer stays within its bounds.

Data structure discovery is easy when symbol tables are available, but very hard when they are not. In the absence of symbol tables, *BinArmor* uses recent research results [29] to reverse engineer the data structures (and especially the buffers) from the binary itself by analyzing memory access patterns (Fig. 1, step i). Something is a struct, if it is accessed like a struct, and an array, if it is accessed like an array. And so on. Next, given the symbols, *BinArmor* dynamically detects buffer accesses (step ii). Finally, in the rewrite stage (step iii), it takes the data structures and the accesses to the buffers, and assigns to each buffer a unique color. Every pointer used to access the buffer for the first time obtains the color of this buffer. *BinArmor* raises an alert whenever, say, a blue pointer accesses a red byte.

Contributions *BinArmor* proactively protects existing C binaries, before we even know whether the code is vulnerable, against attacks on control data *and* non-control data, and it can do so either at object or sub-field granularity. Compared to source-level protection like WIT, *BinArmor* has the advantage that it requires *no access to source code or the original symbol tables*. In addition, in BA-fields mode, by *protecting individual fields* inside a structure rather than aggregates, *BinArmor* is finer-grained than WIT and similar solutions. Also, it prevents overflows on both writes *and* reads, while WIT protects only writes and permits information leakage. Further, we show in Section 9 that points-to analysis (a technique relied on by WIT), is frequently imprecise.

Compared to techniques like taint analysis that also target *binaries*, *BinArmor* detects both control flow and *non-control flow attacks*, whereas taint analysis detects only the former. Also, it detects attacks *immediately* when they occur, rather than sometime later, when a function pointer is used.

The main drawback of *BinArmor* is the very significant slowdown (up to 2.8x for the lighttpd webserver and 1.7x for gzip). While better than most tainting systems (which typically incur 3x-20x), it is much slower than WIT (1.04x for gzip). Realistically, such slowdowns make *BinArmor* in its current form unsuitable for any system that requires high performance. On the other hand, it may be used in application domains where security rather than performance is of prime importance. In addition, because *BinArmor* detects buffer overflows themselves rather than their manifestations, we expect it to be immediately useful for security experts analyzing attacks. Finally, we will show later that we have not explored all opportunities for performance optimization.

Our work builds on dynamic analysis, and thus suffers from the limitations of all dynamic approaches: we can only protect what we execute during the analysis. This work is *not* about code coverage. We rely on existing tools and test suites to cover as much of the binary as possible. Since coverage is never perfect, we may miss buffer accesses and thus incur false negatives. Despite this, *BinArmor* detected all 12 real-world buffer overflow attacks in real-world applications we study (Section 8).

BinArmor takes a conservative approach to prevent false positives (unnecessary program crashes). For instance, no false positives are possible when the protection is limited to structures (BA-objects mode). In BA-fields mode, we can devise scenarios that lead to false positives due to the limited code coverage. However, we did not encounter any in practice, and we will show that they are very unlikely.

Since our dynamic analysis builds on Qemu [6] process emulation which is only available for Linux, we target x86 Linux binaries, generated by `gcc` (albeit of various versions and with different levels of optimization). However, there is nothing fundamental about this and the techniques should apply to other systems also.

2 Some buffer overflows are hard to stop: the Exim attack on non-control data

In December 2010, Sergey Kononenko posted a message on the *exim* developers mailing list about an attack on the *exim* mail server. The news was slashdotted shortly after. The remote root vulnerability in question concerns a heap overflow that causes adjacent heap variables to be overwritten, for instance an access control list (ACL) for the sender of an e-mail message. A compromised ACL

is bad enough, but in *exim* the situation is even worse. Its powerful ACL language can invoke arbitrary Unix processes, giving attackers full control over the machine.

The attack is a typical heap overflow, but what makes it hard to detect is that it does not divert the program's control flow at all. It only overwrites non-control data. ASLR, W⊕X, canaries, system call analysis—all fail to stop or even detect the attack.

Both 'classic' buffer overflows [18], and attacks on non-control data [12] are now mainstream. While attackers still actively use the former (circumventing existing measures), there is simply *no* practical defense against the latter in binaries. Thus, researchers single out non-control data attacks as a serious future threat [30]. *BinArmor* protects against both types of overflows.

3 What to Protect: Buffer Accesses

BinArmor protects binaries by instrumenting buffer accesses to make sure they are safe from overflows. Throughout the paper, *a buffer* is an array that can potentially overflow. Fig. 1 illustrates the general idea, which is intuitively simple: once the program has assigned an array to a pointer, it should not use the same pointer to access elements beyond the array bounds. For this purpose, *BinArmor* assigns colors to arrays and pointers and verifies that the colors of memory and pointer match on each access. After statically rewriting the binary, the resulting code runs natively and incurs overhead only for the instructions that access arrays. In this section, we explain how we obtain buffers and accesses to them when symbols are not available, while Sections 5–7 discuss how we use this information to implement fine-grained protection against buffer overflows.

3.1 Extracting Buffers and Data Structures

Ideally, *BinArmor* obtains information about buffers from the symbol tables. Many projects assume the availability of symbol tables [19, 24]. Indeed, if the binary does come with symbols, *BinArmor* offers very accurate protection. However, as symbols are frequently stripped off in real software, it uses automated reverse engineering techniques to extract them from the binary. *BinArmor* uses a dynamic approach, as static approaches are weak at recovering arrays, but, in principle, they work also [26].

Specifically, we recover arrays using Howard [29], which follows the simple intuition that memory access patterns reveal much about the layout of data structures. In this paper, we sketch only the general idea and refer to the original Howard paper for details [29]. Using binary code coverage techniques [13, 9], Howard executes as many of the execution paths through the binary as possible and observes the memory accesses. To detect arrays, it first detects loops and then treats a memory area as an array if (1) the program accesses the area in a loop (either consecutively, or via arbitrary offsets from the array base), and (2) all accesses 'look like' array accesses (e.g., fixed-size elements). Moreover, it takes into account array accesses outside the loop (including 'first' and 'last' elements), and handles a variety of complications and optimizations (like loop unrolling).

Since arrays are detected dynamically, we should not underestimate the size of arrays, lest we incur false positives. If the array is classified as too small, we might detect an overflow when there is none. In Howard, the data structure extraction is deliberately conservative, so that in practice the size of arrays is either classified exactly right, or overestimated (which never leads to false positives). The reason is that it conservatively extends arrays towards the next variable below or above. Howard is very unlikely to underestimate the array size for compiler-generated code and we never encountered it in any of our tests, although there is no hard guarantee that we never will. Size underestimation is possible, but can happen only if the program accesses the array with multiple base pointers, and behaves consistently and radically different in all analysis runs from the production run.

Over a range of applications, Howard never underestimated an array's size and classified well over 80% of all arrays on the executed paths 'exactly right'—down to the last byte. These arrays represent over 90% of all array bytes. All remaining arrays are either not classified at all or overestimated and thus safe with respect to false positives.

We stressed earlier that Howard aims to err on the safe side, by overestimating the size of arrays to prevent false positives. The question is what the costs are of doing so. Specifically, one may expect an increase in false negatives. While true in theory, this is hardly an issue in practice. The reason is that *BinArmor* only misses buffer overflows that (1) overwrite values immediately following the real array (no byte beyond the (over-)estimation of the array is vulnerable), *and* (2) that overwrite a value that the program did not use separately during the dynamic analysis of the program (otherwise, we would not have classified it as part of the array). Exploitable overflows that satisfy both conditions are rare. For instance, an overflow of a return value would never qualify, as the program always uses the return address separately. Overall, not a single vulnerability in Linux programs for which we could find an exploit qualified.

One final remark about array extraction and false positives; as mentioned earlier, *BinArmor* does not care which method is used to extract arrays and static extractors may be used just as well. However, this is not entirely true. Not underestimating array sizes is crucial.

We consider the problem of finding correct buffer sizes orthogonal to the binary protection mechanism offered by *BinArmor*. Whenever we discuss false positives in *BinArmor*, *we always assume that the sizes of buffers are not underestimated*.

3.2 Instructions to be Instrumented

When *BinArmor* detects buffers to be protected, it dynamically determines the instructions (array accesses), that need instrumenting. The process is straightforward: for each buffer, it dumps all instructions that access it.

Besides accesses, *BinArmor* also dumps all instructions that initialize or manipulate pointers that access arrays.

4 Code Coverage and Modes of Operation

Since *BinArmor* is based on dynamic analysis, it suffers from coverage issues—we can only analyze what we execute. Even the most advanced code coverage tools [9, 13] cover just a limited part of real programs. Lack of coverage causes *BinArmor* to miss arrays and array accesses and thus incur false negatives. Even so, *BinArmor* proved powerful enough to detect *all* attacks we tried (Section 8). What we really want to avoid are false positives: crashes on benign input.

In *BinArmor*, we instrument only those instructions that we encountered during the analysis phase. However, a program path executed at runtime, p_R, may differ from all paths we have seen during analysis A, $\{p_a\}_{a \in A}$, and yet p_R might share parts with (some of) them. Thus, an arbitrary subset of array accesses and pointer manipulations on p_R is instrumented, and as we instrument exactly those instructions that belong to paths in $\{p_a\}_{a \in A}$, it may well happen that we miss a pointer copy, a pointer initialization, or a pointer dereference instruction.

With that in mind, we should limit the color checks performed by *BinArmor* to program paths which use array pointers in ways also seen during analysis. Intuitively, the more scrupulous and fine-grained the color checking policy, the more tightly we need to constrain protected program paths to the ones *seen before*. To address this tradeoff, we offer two modes of *BinArmor* which impose different requirements for the selection of program paths to be instrumented, and offer protection at different granularities: coarse-grained *BA-objects* mode (Section 5), and fine-grained *BA-fields* mode (Section 6).

5 *BA-objects mode*: Object-level Protection

Just like other popular approaches, e.g., WIT [3] and BBC [4], BA-objects mode provides protection at the level of objects used by a program. To do so, *BinArmor* assigns a color to each buffer[1] on stack, heap, or in global memory. Then it makes sure that a pointer to an object of color X never accesses memory of color Y. This way we detect all buffer overflows that aim to overwrite another object in memory.

5.1 What is Permissible? What is not?

Figs. (2.a-2.b) show a function with some local variables, and Fig. (2.c) shows their memory layout and colors. In BA-objects mode, we permit memory accesses within objects, such as the two tick-marked accesses in Fig. (2.c). In the first case, the program perhaps iterates over the elements in the array (at offsets 4, 12, and 20 in the object), and dereferences a pointer to the second element (offset 12) by adding `sizeof(pair_t)` to the array's base pointer at offset 4. In the second case, it accesses the `privileged` field of `mystruct` via a pointer to the last element of the array (offset 24). Although the program accesses a field beyond the array, it remains within the local variable `mystruct`, and (like WIT and other projects), we allow such operations in this mode. Such access patterns commonly occur, e.g., when a `memset()`-like function initializes the entire object.

However, *BinArmor* stops the program from accessing the `len` and `p` fields through a pointer into the structure. `len`, `p` and `mystruct` are separate variables on the stack, and one cannot be accessed through a pointer to the other. Thus, *BinArmor* in BA-objects mode stops inter-object buffer overflow attacks, but not intra-object ones.

5.2 Protection by Color Matching

BinArmor uses colors to enforce protection. It assigns colors to each word of a buffer[1], when the program allocates memory for it in global, heap, or stack memory. Each complete object gets one unique color. All memory which we do not protect gets a unique background color.

When the program assigns a buffer of color X to a pointer, *BinArmor* associates the same color with the register containing the pointer. The color does not change when the pointer value is manipulated (e.g., when the program adds an offset to the pointer), but it is copied when the pointer is copied to a new register. When the pointer is stored to memory, we also store its color to a memory map, to load it later when the pointer is restored.

From now on, *BinArmor* vets each dereference of the pointer to see if it is still in bounds. Vetting pointer dereferences is a matter of checking whether the color of the pointer matches that of the memory to which it points.

Stale Colors and Measures to Rule out False Positives Due to lack of coverage, a program path at runtime may

[1] Or a struct containing the array as this mode operates on objects

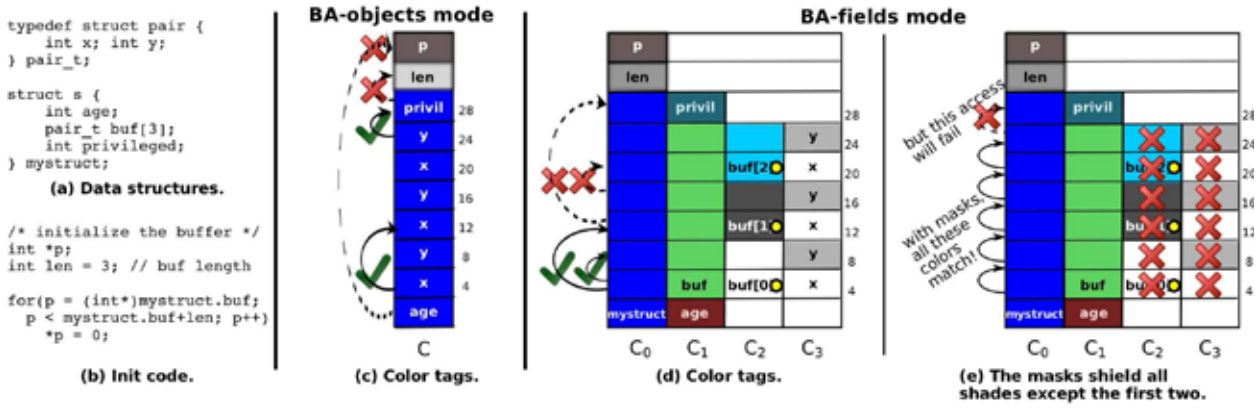


Fig. 2: *BinArmor* colors in BA-objects mode (c) and BA-fields modes (d,e) for sample data structures (a) and code (b).

lack instrumentation on some pointer manipulation instructions. This may lead to the use of a *stale* color.

Consider a function like `memcpy(src,dst)`. Suppose that *BinArmor* misses the `dst` buffer during analysis (it was never used), so that it (erroneously) does not instrument the instructions manipulating the `dst` pointer prior to calling `memcpy()`—say, the instruction that pushes `dst` on the stack. Also suppose that `memcpy()` itself *is* instrumented, so the load of the `dst` pointer into a register obtains the color of that pointer. However, since the original push was not instrumented, *BinArmor* never set that color! If we are lucky, we simply find no color, and everything works fine. If we are unlucky, we pick up a stale color of whatever was previously on the stack at that position[2]. As soon as `memcpy()` dereferences the pointer, the color check fails and the program crashes.

BinArmor removes all false positives of this nature by adding an additional tag to the colors to indicate to which memory address the color corresponds. The tag functions not unlike a tag in a hardware cache entry: to check whether the value we find really corresponds to the address we look for. For instance, if `eax` points to `dst`, the tag contains the address `dst`. If the program copies `eax` to `ebx`, it also copies the color and the tag. When the program manipulates the register (e.g., `eax++`), the tag incurs the same manipulation (e.g., tag_{eax}++). Finally, when the program dereferences the pointer, we check whether the color corresponds to the memory to which the pointer refers. Specifically, *BinArmor* checks the colors on a dereference of `eax`, **iff** *(tag_{eax}==eax)*. Thus, it ignores stale colors and prevents the false positives.

Pointer Subtraction: What if Code is Color Blind? The colors assigned by *BinArmor* prevent a binary from accessing object X though a pointer to object Y. Even though programs in C are not expected to do so, some functions exhibit "color blindness", and directly use a pointer to one object to access another object. The `strcat()` and `__strcpy_chk()` functions in current libc implementations on Linux are the best known examples: to copy a source to a destination string, they access both by the same pointer—adding the *distance* between them to access the remote string.

Our current solution is straightforward. When *BinArmor* detects a pointer subtraction, and later spots when the resultant distance is added to the subtrahend to access the buffer associated with the minuend pointer, it resets the color to reflect the remote buffer, and we protect dereferences in the usual way.

If more complex implementations of this phenomenon appear, we can prevent the associated dereferences from being checked at all. To reach the remote buffer, such scenarios have an operation which involves adding a value derived from the distance between two pointers. *BinArmor* would not include it in the set of instructions to be instrumented, so that the tag of the resultant pointer will not match its value, and the color check will not be performed. False positives are ruled out.

Other projects, like WIT [3] and the pointer analysis-based protection in [5], explicitly assume that a pointer to an object can only be derived from a pointer to the same object. In this sense, our approach is more generic.

5.3 Expect the Unexpected Paths

To justify that *BinArmor* effectively rules out false positives, we have to show that all program paths executed at runtime do not exhibit any false alerts. As we discussed in Section 4, a program path at runtime, p_R, may differ from all paths seen during analysis, while sharing parts with (some of) them. Thus, p_R may ac-

[2]There may be stale colors for the stack value, because it is not practical to clean up all colors whenever memory is no longer in use.

cess an array, while some of the instructions associated with these accesses are not instrumented. The question is whether p_R may cause false positives.

Suppose p_R accesses an array. If `arr` is a pointer to this array, 3 generic types of instruction might be missed, and thus not instrumented by *BinArmor*: (1) an `arr` initialization instruction, (2) an `arr` update/manipulation instruction, and (3) an `arr` dereference instruction.

The crucial feature of *BinArmor* which prevents false positives in cases (1) and (2) are the tags introduced in Section 5.2. They check whether the color associated with a pointer corresponds to the right value. In the case of a pointer initialization or a pointer update instruction missing, the pointer tag does not match its value anymore, its color is considered invalid, and it is not checked on dereferences. Finally, if an `arr` dereference instruction is not instrumented, it only means that the color check is not performed. Again, it can only result in false negatives, but never false positives.

6 *BA-fields mode*: a Colorful Armor

BA-objects mode and BA-fields mode differ significantly in the granularity of protection. Where BA-objects mode protects memory at the level of objects, BA-fields mode offers finer-grained protection—at the level of fields in structures. Thus, BinArmor in BA-fields mode stops not only inter-object buffer overflow attacks, but also intra-object ones. We shall see, the extra protection increases the chances of false positives which should be curbed.

6.1 What is Permissible? What is not?

Consider the structure in Fig. (2.a) with a memory layout as shown in Fig. (2.d). Just like in BA-objects mode, *BinArmor* now also permits legitimate memory accesses such as the two tick-marked accesses in Fig. (2.d).

But unlike in BA-objects mode, *BinArmor* in BA-fields mode stops the program from accessing the `privileged` field *via* a pointer into the array. Similarly, it prevents accessing the `x` field in one array element from the `y` field in another. Such accesses that do not normally occur in programs are often symptomatic of attacks[3].

6.2 Shaded Colors

BinArmor uses a *shaded* color scheme to enforce fine-grained protection. Compared to BA-objects mode, the color scheme used here is much richer. In Section 5, the whole object was given a single color, but in BA-fields mode, we add shades of colors to distinguish between individual fields in a structure. First, we sketch how we assign the colors. Next, we explain how they are used.

Since *BinArmor* knows the structure of an object to be protected, it can assign separate colors to each variable and to each field. The colors are hierarchical, much like real colors: lime green is a shade of green, and dark lime green and light lime green, are gradations of lime green, etc. Thus, we identify a byte's color as a sequence of shades: $C_0 : C_1 : .. : C_N$, where we interpret C_{i+1} as a shade of color C_i. Each shade corresponds to a nesting level in the data structure. This is illustrated in Fig. (2.d).

The base color, C_0, corresponds to the complete object, and is just like the color used by *BinArmor* in BA-objects mode. It distinguishes between individually allocated objects. At level 1, the object in Fig. (2.d) has three fields, each of which gets a unique shade C_1. The two integer fields do not have any further nesting, but the array field has two more levels: array elements and fields within the array elements. Again, we assign a unique shade to each array element and, within each array element, to each field. The only exceptions are the base of the array and the base of the structs—they remain blank for reasons we explain shortly. Finally, each color C_i has a type flag indicating whether it is an array element shown in the figure as a dot (a small circle on the right).

We continue the coloring process, until we reach the maximum nesting level (in the figure, this happens at C_3), or exhaust the maximum color depth N. In the latter case, the object has more levels of nesting than *BinArmor* can accommodate in shades, so that some of the levels will collapse into one, 'flattening' the substructure. Collapsed structures reduce *BinArmor*'s granularity, but do not cause problems otherwise. In fact, most existing solutions (like WIT [3] and BBC [4]) operate only at the granularity of the full object.

Protection by Color Matching The main difference between the color schemes implemented in BA-objects mode and BA-fields mode is that colors are more complex now and include multiple shades. We need a new procedure to compare them, and decide what is legal.

The description of the procedure starts in exactly the same way as in BA-objects mode. When a buffer of color X is assigned to a pointer, *BinArmor* associates the same color with the register containing the pointer. The color does not change when the pointer value is manipulated (e.g., when the program adds an offset to the pointer), but it is copied when the pointer is copied to a new register. When the program stores a pointer to memory, we also store its color to a memory map, to load it later when the pointer is restored to a register.

The difference from the BA-objects mode is in the color update rule: when the program dereferences a register, we update its color so that it now corresponds to

[3] Note: if they *do* occur, either `Howard` classifies the data structures differently, or *BinArmor* detects these accesses in the analysis phase, and applies *masks* (Section 6.2), so they do not cause problems.

the memory location associated with the register. The intuition is that we do not update colors on intermediate pointer arithmetic operations, but that the colors represent pointers used by the program to access memory.

From now on, *BinArmor* vets each dereference of the pointer to see if it is still in bounds. Vetting pointer dereferences is a matter of checking whether the color of the pointer matches that of the memory it points to—in all the shades, from left to right. Blank shades serve as wild cards and match any color. Thus, leaving bases of structures and arrays blank guarantees that a pointer to them can access all internal fields of the object.

Finally, we handle the common case where a pointer to an array element derives from a pointer to another element of the array. Since array elements in Fig. (2c) differ in C_2, such accesses would normally not be allowed, but the dots distinguish array elements from structure fields. Thus we are able to grant these accesses. We now illustrate these mechanisms for our earlier examples.

Suppose the program has already accessed the first array element by means of a pointer to the base of the array at offset 4 in the object. In that case, the pointer's initial color is set to C_1 of the array's base. Next, the program adds `sizeof(pair_t)` to the array's base pointer and dereferences the result to access the second array element. At that point, *BinArmor* checks whether the colors match. C_0 clearly matches, and since the pointer has only the C_1 color of the first array element, its color and that of the second array element match. Our second example, accessing the `y` field from the base of the array, matches for the same reason.

However, an attacker cannot use this base pointer to access the `privileged` field, because the C_1 colors do not match. Similarly, going from the `y` field in the second array element to the `x` field in the third element will fail, because the C_2 shades differ.

The Use of Masks: What if Code is Color Blind? Programs do not always access data structures in a way that reflects the structure. They frequently use functions similar to `memset` to initialize (or copy) an entire object, with all subfields and arrays in it. Unfortunately, these functions do not heed the structure at all. Rather, they trample over the entire data structure in, say, word-size strides. Here is an example. Suppose `p` is a pointer to an integer and we have a custom `memset`-like function:

```
for (p=objptr, p<sizeof(*objptr); p++) *p = 0;
```

The code is clearly 'color blind', but while it violates the color protection, *BinArmor* should not raise an alert as the accesses are all legitimate. But it should not ignore color blindness either. For instance, the initialization of one object should not trample over *other* objects. Or inside the structure of Fig. (2.b): an initialization of the array should not overwrite the `privileged` field.

One (bad) way to handle such color blindness is to white-list the code. For instance, we could ignore all accesses from white-listed functions. While this helps against some false alerts, it is not a good solution for two reasons. First, it does not scale; it helps only against a few well-known functions (e.g., libc functions), but not against applications that use custom functions to achieve the same. Second, as it ignores these functions altogether, it would miss attacks that use this code. For instance, the initialization of (just) the buffer could overflow into the privilege field.

Instead, *BinArmor* exploits the shaded colors of Section 6.2 to implement *masks*. Masks shield code that is color blind from some of the structure's subtler shades. For instance, when the initialization code in Fig. (2.b) is applied to the array, we filter out all shades beyond C_1: the code is then free to write over all the records in the array, but cannot write beyond the array. Similarly, if an initialization routine writes over the entire object, we filter all shades except C_0, limiting all writes to this object.

Fig. (2.e) illustrates the usage of masks. The code on the left initializes the array in the structure of Fig. 2. By masking all colors beyond C_0 and C_1, all normal initialization code is permitted. If attackers can somehow manipulate the `len` variable, they could try to overflow the buffer and change the `privileged` value. However, in that case the C_1 colors do not match, and *BinArmor* will abort the program.

To determine whether a memory access needs masks (and if so, what sort), *BinArmor*'s dynamic analysis first marks all instructions that trample over multiple data structures as 'color blind' and determines the appropriate mask. For instance, if an instruction accesses the base of the object, *BinArmor* sets the masks to block out all colors except C_0. If an instruction accesses a field at the k^{th} level in the structure, *BinArmor* sets the masks to block out all colors except $C_0 ... C_k$. And so on.

Finding the right masks to apply and the right places to do so, requires fairly subtle analysis. *BinArmor* needs to decide *at runtime* which part of the shaded color to mask. In the above example, if the program initializes the whole structure, *BinArmor* sets the masks to block out all colors except C_0. If the same function is called to initialize the array, however, only C_2 and C_3 are shielded. To do so, *BinArmor*'s dynamic analysis tracks the *source* of the pointer used in the 'color blind' instruction, i.e., the base of the structure or array. The instrumentation then allows for accesses to all fields included in the structure (or substructure) rooted at this source. Observe that not all such instructions need masks. For instance, code that zeros all words in the object by adding increasing offsets to the *base* of the object, has no need for masks. After all, be-

cause of the blank shades the base of the object permits access to the entire object even without masks.

BinArmor enforces the masks when rewriting the binary. Rather than checking all shades, it checks only the instructions' *visible* colors for these instructions.

Pointer Subtraction As discussed in Section 5.2, some functions exhibit color blindness, and use a pointer to one object to access another. Both the problem and its solution are exactly the same as for BA-fields mode.

6.3 Why We do Not See False Positives

Given an accurate or conservative estimate of array sizes, the only potential cause of false positives is lack of coverage. As explained in Section 5, we do not address the array size underestimation here–we simply require either symbol tables or a conservative data structure extractor (Section 3). But other coverage issues occur regardless of symbol table availability and must be curbed.

Stale Colors and Tags In Section 5.2, we showed that lack of coverage could lead to the use of stale colors in BA-objects mode. Again, the problem and its solution are the same as for BA-fields mode.

Missed Masks and Context Checks Limited code coverage may also cause *BinArmor* to miss the *need* for masks and, unless prevented, lead to false positives. Consider again the example custom `memset` function of Section 6.2. The code is color blind, unaware of the underlying data structure, and accesses the memory according to its own pattern. To prevent false positives, we introduced *masks* that filter out some shades to allow for benign memory accesses.

Suppose that during analysis the custom `memset` function is invoked only once, to initialize an array of 4-byte fields. No masks are necessary. Later, in a production run, the program takes a previously unknown code path, and uses the same function to access an array of 16-byte structures. Since it did not assign masks to this function originally, *BinArmor* now raises a (false) alarm.

To prevent the false alarm, we keep two versions of each function in a program: a vanilla one, and an instrumented one which performs color checking. When the program calls a function, *BinArmor* checks whether the function call also occurred at the same point during the analysis by examining the call stack. (In practice, we take the top 3 function calls.) Then, it decides whether or not to execute the instrumented version of the function. It performs color checking only for layouts of data structures we *have seen before*, so we do not encounter code that accesses memory in an unexpected way.

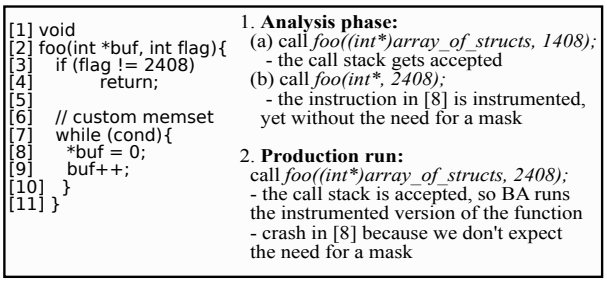


Fig. 3: BA-fields mode: a scenario leading to false positives.

6.4 Are False Positives Still Possible?

While the extra mechanism to prevent false positives based on context checks is effective in practice, it does not give any strong guarantees. The problem is that a call stack does not identify the execution context with absolute precision. Fig. 3 shows a possible problematic scenario. In this case, it should not be the call stack, but a node in the program control flow graph which identifies the context. Only if we saw the loop in lines [6-9] initializing the `array_of_structs`, should we allow for an instrumented version of it at runtime. Observe that the scenario is fairly improbable. First, the offensive function must exhibit the need for masks, that is, it must access subsequent memory locations through a pointer to a previous field. Second, it needs to be called twice with very particular sets of arguments before it can lead to the awkward situation.

As we did not encounter false positives in *any* of our experiments, and BA-fields mode offers powerful, fine-grained protection, we think that the risk may be acceptable in application domains that can handle rare crashes.

7 Efficient Implementation

Protection by color matching combined with masks for color blindness allows *BinArmor* to protect data structures at a finer granularity than previous approaches. Even so, the mechanisms are sufficiently simple to allow for efficient implementations. *BinArmor* is designed to instrument 32-bit ELF binaries for the Linux/x86 platforms. Like Pebil [21], it performs static instrumentation, i.e., it inserts additional code and data into an executable, and generates a new binary with permanent modifications. We first describe how *BinArmor* modifies the layout of a binary, and next present some details of the instrumentation. (For a full explanation refer to [32].)

7.1 Updated Layout of ELF Binary

To accommodate new code and data required by the instrumentation, *BinArmor* modifies the layout of an ELF binary. The original data segment stays unaltered,

while we modify the text segment only in a minor way—just to allow for the selection of the appropriate version of a function (Section 6.3), and to assure that the resulting code works correctly—mainly by adjusting jump targets to reflect addresses in the updated code (refer to Section 7.2). To provide protection, *BinArmor* inserts a few additional segments in the binary: BA_Initialized_Data, BA_Uninitialized_Data, BA_Procedures, and BA_Code.

Both data segments–*BA_(Un)Initialized_Data*– store data structures that are internally used by *BinArmor*, e.g., data structures color maps, or arrays mapping addresses in the new version of a binary to the original ones. The *BA_Procedures* code segment contains various chunks of machine code used by instrumentation snippets (e.g., a procedure that compares the color of a pointer with the color of a memory location). Finally, the *BA_Code* segment is the pivot of the *BinArmor* protection mechanism—it contains the original program's functions instrumented to perform color checking.

7.2 Instrumentation Code

To harden the binary, we rewrite it to add instrumentation to those instructions that dereference the array pointers. In BA-fields mode, we use multi-shade colors only if the data structures are nested. When we can tell that a variable is a string, or some other non-nested array, we switch to a simpler, single-level color check.

To provide protection, *BinArmor* reorganizes code at the instruction level. We do not need to know function boundaries, as we instrument instructions which were classified as array accesses, along with pointer move or pointer initialization instructions, during the dynamic analysis phase. We briefly describe the main steps taken by *BinArmor*: (1) inserting trampolines and method selector, (2) code relocation, (3) inserting instrumentation.

Inserting trampolines and method selector. The role of a *method selector* is to decide whether a vanilla or an instrumented function should be executed (see Section 6.3), and then jump to it. In *BinArmor*, we place a *trampoline* at the beginning of each (dynamically detected) function in the original text segment, which jumps to the method selector. The selector picks the right code to continue execution, as discussed previously.

Code relocation. *BinArmor*'s instrumentation framework must be able to add an arbitrary amount of extra code between any two instructions. In turn, targets of *all* `jump` and `call` instructions in a binary need to be adjusted to reflect new values of the corresponding addresses. As far as direct/relative jumps are concerned, we simply calculate new target addresses, and modify the instructions. Our solution to indirect jumps is similar to [31]: they are resolved at runtime, by using arrays maintaining a mapping between old and new addresses.

```
_dereference_check_start:
# check whether tag value matches the pointer
cmp %edx, register_tag_edx
jne _dereference_check_end
[save %eax and %ebx used in instrumentation]
lea (%edx, %eax, 4), %ebx
call get_color_of_ebx ; loaded to %bx
mov register_color_edx, %ax
call color_match ; compare colors in %ax and %bx
cmpl $0, %eax    ; check result
je _dereference_ok

_dereference_bad:
["crash"]

_dereference_ok:
[restore %eax and %ebx used in instrumentation]

_dereference_check_end:
movl $0x1234, (%edx, %eax, 4); execute original instr
```

Fig. 4: Instrumentation for an array pointer dereference (with 16b colors and tags). The original instruction is `mov 0x1234,(%edx,%eax,4)`. We replace it by code similar to that presented in the figure (but more efficient).

Inserting instrumentation. Snippets come in many shapes. For instance, snippets to handle pointer dereferences, to handle pointer move instructions, or to color memory returned by `malloc`. Some instructions require that a snippet performs more than one action. For example, an instruction which stores a pointer into an array, needs to both store the color of the pointer in a color map, and make sure that the store operation is legal. For an efficient instrumentation, we have developed a large number of carefully tailored snippets.

Colors map naturally on a sequence of small numbers. For instance, each byte in a 32-bit or 64-bit word may represent a shade, for a maximum nesting level of 4 or 8, respectively. Doing so naively, incurs a substantial overhead in memory space, but, just like in paging, we need only allocate memory for color tags when needed. The same memory optimization is often used in dynamic taint analysis. In the implementation evaluated in Section 8, we use 32 bit colors with four shades and 16 bit tags.

Fig. 4 shows an example of an array dereference in the binary hardened by *BinArmor*. The code is simplified for brevity, but otherwise correct. We see that each array dereference incurs some extra instructions. If the colors do not match, the system crashes. Otherwise, the dereference executes as intended. We stress that the real implementation is more efficient. For instance, adding two `call` instructions would be extremely expensive. In reality, *BinArmor* uses code snippets tailored to performance

8 Evaluation

We evaluate *BinArmor* on performance and on effectiveness in stopping attacks. As the analysis engine is based on the Qemu processor emulation, which is currently only available on Linux, all examples are Linux-based. However, the approach is not specific to any operating system.

We have performed our analysis for binaries compiled with two compiler versions, `gcc-3.4` and `gcc-4.4`, and with different optimization levels. All results presented in this section are for binaries compiled with `gcc-4.4 -O2` and without symbols, i.e., completely stripped. We reconstruct the symbols using Howard [29].

Performance To evaluate the performance of *BinArmor* operating in BA-fields mode[4], we compare the speed of instrumented (armored) binaries with that of unmodified implementations. Our test platform is a Linux 2.6 system with an Intel(R) Core(TM)2 Duo CPU clocked at 2.4GHz with 3072KB L2 cache. The system has 4GB of memory. For our experiments we used an Ubuntu 10.10 install. We ran each test multiple times and present the median. Across all experiments, the 90th percentiles were typically within 10% and never more than 20% off the mean.

We evaluate the performance of *BinArmor* with a variety of applications—all of the well-known *nbench* integer benchmarks [1]—and a range of real-world programs. We picked the *nbench* benchmark suite, because it is compute-intensive and several of the tests should represent close to worst-case scenarios for *BinArmor*.

For the real-world applications, we chose a variety of very different programs: a network server (`lighttpd`), several network clients (`wget`, `htget`), and a more compute-intensive task (`gzip`). `Lighttpd` is a high-performance web server used by such popular sites as YouTube, SourceForge, Wikimedia, Meebo, and ThePirateBay. `Wget` and `htget` are well-known command-line web clients. `Gzip` implements the DEFLATE algorithm which includes many array and integer operations.

Fig. 5 shows that for real I/O-intensive client-side applications like `wget` and `htget` the slowdown is negligible, while `gzip` incurs a slow-down of approximately 1.7x. As `gzip` is a very expensive test for *BinArmor*, the slow-down was less than we expected. The overhead for a production-grade web server like `lighttpd` is also low: less than 2.8x for all object sizes, and as little as 16% for large objects. In networking applications I/O dominates the performance and the overhead of *BinArmor* is less important.

[4]The reason is that BA-fields mode is the most fine-grained, although in practice, the performance of BA-objects mode is very similar.

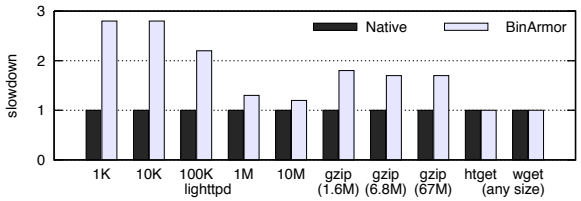


Fig. 5: Performance overhead for real world applications: *lighttpd* – for 5 object sizes (in connections/s as measured by *httperf*), *gzip* – for 3 object sizes, *htget* and *wget*.

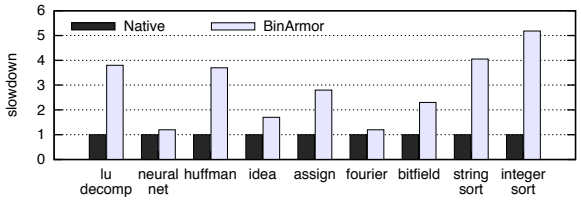


Fig. 6: Performance overhead for the compute-intensive *nbench* benchmark suite.

Fig. 6 shows the results for the very compute-intensive *nbench* test suite. The overall slowdown for nbench is 2.9x. Since this benchmark suite was chosen as worst-case scenario and we have not yet fully optimized *BinArmor*, these results are quite good. Some of the tests incurred a fairly minimal slow-down. Presumably, these benchmarks are dominated by operations other than array accesses. String sort and integer sort, on the other hand, manipulate strings and arrays constantly and thus incur much higher slowdowns. They really represent the worst cases for *BinArmor*.

Effectiveness Table 1 shows the effectiveness of *BinArmor* in detecting attacks on a range of real-life software vulnerabilities. Specifically, these attacks represent *all* vulnerabilities on Linux programs for which we found working exploits. *BinArmor* operating in either mode detected all attacks we tried and did not generate any false positives during any of our experiments. The attacks detected vary in nature and include overflows on both heap and stack, local and remote, and of both control and non-control data.

The detection of attacks on non-control data is especially encouraging. While control flow diversions would trigger alerts also on taint analysis systems like Argos [25] and Minemu [8], to the best of our knowledge, no other security measure for stripped binaries would be able to detect such attacks. As mentioned earlier, security experts expect non-control data attacks to become even more important attack vectors in the near future [12, 30].

Application	Vulnerability type	Security advisory
Aeon 0.2a	Stack overflow	CVE-2005-1019
Aspell 0.50.5	Stack overflow	CVE-2004-0548
Htget 0.93 (1)	Stack overflow	CVE-2004-0852
Htget 0.93 (2)	Stack overflow	
Iwconfig v.26	Stack overflow	CVE-2003-0947
Ncompress 4.2.4	Stack overflow	CVE-2001-1413
Proftpd 1.3.3a	Stack overflow	CVE-2010-4221
bc-1.06 (1)	Heap overflow	Bugbench [22]
bc-1.06 (2)	Heap overflow	Bugbench [22]
Exim 4.41	Heap overflow*	CVE-2010-4344
Nullhttpd-0.5.1	Heap overflow†	CVE-2002-1496
Squid-2.3	Heap overflow†	Bugbench [22]

* A non-control-diverting attack. † A reproduced attack.

Table 1: Tested vulnerabilities: *all attacks were stopped by BinArmor*, including the attack on non-control data.

9 Related Work

The easiest way to prevent memory corruptions is to do so at the source level, using a safe language or compiler extension. However, as access to source code or recompilation is often not an option, many binaries are left unprotected. Our work is about protecting binaries. Since it was inspired by the WIT compiler extension, we briefly look at compile time solutions also.

Protection at compile time Managed languages like Java and C# are safe from buffer overflows by design. Cyclone [20] and CCured [14] show that similar protection also fits dialects of C—although the overhead is not always negligible. Better still, data flow integrity (DFI) [10], write integrity testing (WIT) [3], and baggy bounds checking (BBC) [4] are powerful protection approaches against memory corruptions for unmodified C.

BinArmor was inspired by the WIT compiler extension—a defense framework that marries immediate (fail-stop) detection of memory corruption to excellent performance. WIT assigns a color to each object in memory and to each write instruction in the program, so that the color of memory always matches the color of an instruction writing it. Thus all buffers which can be potentially accessed by the same instruction share the same color. WIT employs points-to analysis to find the set of objects written by each instruction. If several objects share the same color, WIT might fail to detect attacks that use a pointer to one object to write to the other. To get a grasp of the precision, we implemented points-to analysis ourselves, and applied it to global arrays in `gzip-1.4`. Out of 270 buffers, 124 have a unique color, and there are two big sets of objects that need to share it: containing 64 and 68 elements. (We assume that we provide templates for the `libc` functions. Otherwise, the precision is worse.) To deal with these problems, WIT additionally inserts small guards between objects, which cannot be written by any instruction. They provide an extra protection against sequential buffer overflows. *BinArmor* tracks colors of objects dynamically, so each array is assigned a unique color.

Also, WIT and BBC protect at the granularity of memory allocations. If a program allocates a structure that contains an array as well as other fields, overflows within the structure go unnoticed. As a result, the attack surface for memory attacks is still huge. SoftBound is one of the first tools to protect subfields in C structures [23]. Again, SoftBound requires access to source code.

BinArmor's protection resembles that of WIT, but without requiring source code, debugging information, or even symbol tables. Unlike WIT, it protects at the granularity of subfields in C `structs`. It prevents not just out-of-bounds writes, as WIT does, but also reads. As a drawback, *BinArmor* may be less accurate, since dynamic analysis may not cover the entire program.

Protection of binaries Arguably some of the most popular measures to protect against memory corruption are memory debuggers like Purify and Valgrind [24]. These powerful testing tools are capable of finding many memory errors without source code. However, they incur overheads of an order of magnitude or more. Moreover, their accuracy depends largely on the presence of debug information and symbol tables. In contrast, *BinArmor* is much faster and requires neither.

One of the most advanced approaches to binary protection is XFI [19]. Like memory debuggers, XFI requires symbol tables. Unlike memory debuggers, DTA, or *BinArmor*, XFI's main purpose is to protect host software that loads modules (drivers in the kernel, OS processes, or browser modules) and it requires explicit support from the hosting software–to grant the modules access to a slice of the address space. It offers protection by a combination of control flow integrity, stack splitting, and memory access guards. Memory protection is at the granularity of the module, and for some instructions, the function frame. The memory guards will miss most overflows that modify non-control data.

An important class of approaches to detect the *effects* of memory corruption attacks is based on dynamic taint analysis (DTA) [15]. DTA does not detect the memory corruption itself, but may detect malicious control flow transfers. Unfortunately, the control flow transfer occurs at a (often much) later stage. With typical slowdowns of an order of magnitude, DTA in software is also simply *too expensive* for production systems.

Non-control data attacks are much harder to stop [12]. [11] pioneered an interesting form of DTA to detect some of these attacks: pointers become tainted if their values

are influenced by user input, and an alert is raised if a tainted value is dereferenced. However, pointer taintedness for detecting non-control data attacks is shown to be impractical for complex architectures like the x86 and popular operating systems [28]. The problems range from handling table lookups to implicit flows and result in false positives and negatives. Moreover, by definition, pointer taintedness cannot detect attacks that do not dereference a tainted pointer, such as an attack that would overwrite the `privileged` field in Fig. (2a).

10 Discussion

Obviously, *BinArmor* is not flawless. In this section, we discuss some generic limitations.

With a dynamic approach, *BinArmor* protects only arrays detected by Howard. If the attackers overflow other arrays, we will miss the attacks. Also, if particular array accesses are not exercised in the analysis phase, the corresponding instructions are not instrumented either. Combined with the tags (Section 5.2), this lack of accuracy can only cause false negatives, but never false positives. In practice, as we have seen in Section 8, *BinArmor* was able to protect all vulnerable programs we tried.

Howard itself is designed to err on the safe side. In case of doubt, it overestimates the size of an array. Again, this can lead to false negatives, but not false positives. However, if the code is strongly obfuscated or deliberately designed to confuse Howard, we do not guarantee that it will never misclassify a data structure in such a way that it will cause a false positive. Still, it is unlikely, because to do so, the behavior during analysis should also be significantly different from that during the production run. In our view, the risk is acceptable for software deployments that can tolerate rare crashes.

We have implemented two versions of *BinArmor*: BA-objects mode, and BA-fields mode. While the latter protects memory at a fine-grained granularity, there exist theoretical situations that can lead to false alerts. However, in practice we did not encounter any problems. Since the protection offered is very attractive — *BinArmor* protects individual fields within structures — we again think that the risk is acceptable.

Code coverage is a limitation of all dynamic analysis techniques and we do not claim any contribution to this field. Interestingly, code coverage can also be 'too good'. For instance, if we were to trigger a buffer overflow during the analysis run, *BinArmor* would interpret it as normal code behavior and not prevent similar overruns during production. Since coverage techniques to handle complex applications are currently still fledgling, this is mostly an academic problem. At any rate, if binary code coverage techniques are so good as to find such real problems in the testing phase, this can only be beneficial for the quality of software.

11 Future work

BinArmor's two main problems are accuracy (in terms of false negatives and false positives) and performance (in terms of the slowdown of the rewritten binary). In this section, we discuss ways to address these problems.

First, the root cause of *BinArmor*'s false negative and false positive problems is the lack of code coverage. Our next target, therefore, is to extend the paths covered dynamically by means of static analysis. For instance, we can statically analyze the full control flow graphs of all functions called at runtime. Static analysis in general is quite hard, due to indirect calls and jumps, but within a single function indirect jumps are often tractable (they are typically the result of `switch` statements that are relatively easy to handle).

Second, the cause of *BinArmor*'s slowdown is the instrumentation that adds overhead to every access to an array that *BinArmor* discovered. We can decrease the overhead using techniques that are similar to those applied in WIT. For instance, there is no need to perform checks on instructions which calculate the address to be dereferenced in a deterministic way, say at offset 0x10 from a base pointer. Thus, our next step is to analyze the instructions that are candidates for instrumentation and determine whether the instrumentation is strictly needed.

12 Conclusions

We described a novel approach to harden binary software proactively against buffer overflows, without access to source or symbol tables. Besides attacks that divert the control flow, we also detect attacks against non-control data. Further, we demonstrated that our approach stops a variety of real exploits. Finally, as long as we are conservative in classifying data structures in the binaries, our method will not have false positives. On the downside, the overhead of our approach in its current form is quite high—making it unsuitable for many application domains today. However, we also showed that significant performance optimizations may still be possible. It is our view that protection at the binary level is important for dealing with real threats to real and deployed information systems.

Acknowledgements

This work is supported by the European Research Council through project ERC-2010-StG 259108-ROSETTA and the EU FP7 SysSec Network of Excellence. The authors are grateful to David Brumley and his team for pro-

viding us with several local exploits, and to Erik Bosman and Philip Homburg for their work on the Exim exploit.

References

[1] BYTE Magazine nbench benchmark. `http://www.tux.org/~mayer/linux/bmark.html`.

[2] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-Flow integrity. In *Proc. of the 12th ACM conference on Computer and communications security* (2005), CCS'05.

[3] AKRITIDIS, P., CADAR, C., RAICIU, C., COSTA, M., AND CASTRO, M. Preventing memory error exploits with WIT. In *Proc. of the IEEE Symposium on Security and Privacy*, S&P'08.

[4] AKRITIDIS, P., COSTA, M., CASTRO, M., AND HAND, S. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proc. of the 18th Usenix Security Symposium* (2009), USENIX-SS'09.

[5] AVOTS, D., DALTON, M., LIVSHITS, V. B., AND LAM, M. S. Improving software security with a C pointer analysis. In *Proc. of the 27th Intern. Conf. on Software Engineering (ICSE)* (2005).

[6] BELLARD, F. QEMU, a fast and portable dynamic translator. In *Proc. of USENIX 2005 Annual Technical Conference*, ATEC '05.

[7] BHATKAR, S., DUVARNEY, D. C., AND SEKAR, R. Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *Proc. of the 12th conference on USENIX Security Symposium* (2003), SSYM'03.

[8] BOSMAN, E., SLOWINSKA, A., AND BOS, H. Minemu: The Worlds Fastest Taint Tracker. In *Proc. of 14th International Symposium on Recent Advances in Intrusion Detection* (2011), RAID 2011.

[9] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc.s of the 8th USENIX Symposium on Operating Systems Design and Implementation* (2008), OSDI'08.

[10] CASTRO, M., COSTA, M., AND HARRIS, T. Securing software by enforcing data-flow integrity. In *Proc. of the 7th USENIX Symp. on Operating Systems Design and Impl.* (2006), OSDI'06.

[11] CHEN, S., XU, J., NAKKA, N., KALBARCZYK, Z., AND IYER, R. K. Defeating memory corruption attacks via pointer taintedness detection. In *Proc. of the 2005 International Conference on Dependable Systems and Networks* (2005), DSN '05.

[12] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-control-data attacks are realistic threats. In *Proc. of 14th USENIX Security Symposium* (2005), SSYM'05.

[13] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2E: A platform for in vivo multi-path analysis of software systems. In *Proc. of 16th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011).

[14] CONDIT, J., HARREN, M., MCPEAK, S., NECULA, G. C., AND WEIMER, W. CCured in the real world. In *Proc of the 2003 Conf. on Programming languages design and implementation*, POPL'03.

[15] COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A., ZHOU, L., ZHANG, L., AND BARHAM, P. Vigilante: end-to-end containment of internet worms. In *Proc. of the 20th ACM Symposium on Operating Systems Principles* (2005), SOSP'05.

[16] COWAN, C., PU, C., MAIER, D., HINTONY, H., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proc. of the 7th USENIX Security Symposium* (1998), SSYM'98.

[17] CWE/SANS. TOP 25 Most Dangerous Software Errors. `www.sans.org/top25-software-errors`, 2011.

[18] ELIAS LEVY (ALEPH ONE). Smashing the stack for fun and profit. *Phrack 7*, 49 (1996).

[19] ERLINGSSON, U., VALLEY, S., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. XFI: software guards for system address spaces. In *Proc. of the 7th USENIX Symp. on Operating Systems Design and Implementation* (2006), OSDI '06.

[20] JIM, T., MORRISETT, G., GROSSMAN, D., HICKS, M., CHENEY, J., AND WANG, Y. Cyclone: A safe dialect of C. In *USENIX 2002 Annual Technical Conference*, ATEC '02.

[21] LAURENZANO, M., TIKIR, M. M., CARRINGTON, L., AND SNAVELY, A. PEBIL: Efficient static binary instrumentation for Linux. In *Proc. of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS-2010.

[22] LU, S., LI, Z., QIN, F., TAN, L., ZHOU, P., AND ZHOU, Y. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools* (2005).

[23] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., AND ZDANCEWIC, S. SoftBound: highly compatible and complete spatial memory safety for C. In *Proc. of PLDI'09*.

[24] NETHERCOTE, N., AND SEWARD, J. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proc. of the 3rd Intern. Conf. on Virtual Execution Environ.* (2007), VEE.

[25] PORTOKALIDIS, G., SLOWINSKA, A., AND BOS, H. Argos: an Emulator for Fingerprinting Zero-Day Attacks. In *Proc. of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems* (2006), EuroSys '06.

[26] REPS, T., AND BALAKRISHNAN, G. Improved memory-access analysis for x86 executables. In *CC'08/ETAPS'08: Proc. of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction* (2008).

[27] SLOWINSKA, A., AND BOS, H. The Age of Data: Pinpointing Guilty Bytes in Polymorphic Buffer Overflows on Heap or Stack. In *Proc. of the 23rd Annual Computer Security Applications Conference* (2007), ACSAC'07.

[28] SLOWINSKA, A., AND BOS, H. Pointless tainting?: evaluating the practicality of pointer tainting. In *EuroSys '09: Proc. of the 4th ACM European conf. on Computer systems* (2009).

[29] SLOWINSKA, A., STANCESCU, T., AND BOS, H. Howard: a dynamic excavator for reverse engineering data structures. In *Proceedings of NDSS* (2011).

[30] SOTIROV, A. Modern exploitation and memory protection bypasses. USENIX Security invited talk, `www.usenix.org/events/sec09/tech/slides/sotirov.pdf`, August 2009.

[31] SRIDHAR, S., SHAPIRO, J. S., AND NORTHUP, E. HDTrans: An open source, low-level dynamic instrumentation system. In *Proc. of the 2nd Intern. Conf. on Virtual Execution Environ.* (2006).

[32] STANCESCU, T. BodyArmor: Adding Data Protection to Binary Executables. Master's thesis, VU Amsterdam, 2011.

[33] TEAM, P. Design and implementation of PAGEEXEC. `http://pax.grsecurity.net/docs/pageexec.old.txt`, November 2000.

Abstractions for Usable Information Flow Control in Aeolus

Winnie Cheng[*] Dan R. K. Ports David Schultz
Victoria Popic[†] Aaron Blankstein[‡] James Cowling
Dorothy Curtis Liuba Shrira[§] Barbara Liskov

MIT CSAIL [*]*IBM Research* [†]*Stanford* [‡]*Princeton* [§]*Brandeis*

Abstract

Despite the increasing importance of protecting confidential data, building secure software remains as challenging as ever. This paper describes *Aeolus*, a new platform for building secure distributed applications. Aeolus uses information flow control to provide confidentiality and data integrity. It differs from previous information flow control systems in a way that we believe makes it easier to understand and use. Aeolus uses a new, simpler security model, the first to combine a standard principal-based scheme for authority management with thread-granularity information flow tracking. The principal hierarchy matches the way developers already reason about authority and access control, and the coarse-grained information flow tracking eases the task of defining a program's security restrictions. In addition, Aeolus provides a number of new mechanisms (authority closures, compound tags, boxes, and shared volatile state) that support common design patterns in secure application design.

1 Introduction

Confidential information, such as credit card numbers and medical records, is increasingly stored online. Keeping this information secure despite malicious attacks and human errors is a high priority, as evidenced by recent regulatory requirements [8, 11]. Building secure software, however, remains as challenging as ever.

Information flow control offers a promising option for construction of secure software. Traditionally, information has been secured through access control, which constrains who is allowed to read and write information. Information flow control complements this by allowing an untrusted entity access to sensitive data as long as it does not reveal the data. It has long been of interest in military systems [21], where having access to "top secret" information does not imply the information can be released to an "unclassified" user.

The *decentralized* information flow control (DIFC) model [15] generalizes the approach and makes it useful for arbitrary applications, by replacing central control with the ability for individuals to define restrictions on the use of their own information. However, despite much recent research on DIFC systems [6, 9, 15–17, 23, 24], information flow control has not been widely used in practice. We believe this is because there are several requirements that are not met effectively by existing systems:

1. Developers require an understandable and flexible authority structure. DIFC depends on the use of authority to determine whether information can be released or trusted. Programmers who use DIFC must be able to understand the authority structure their applications depend on, and they must be able to change this structure, both by establishing new lines of authority and by revoking existing authority.
2. Developers require support for the principle of least privilege, to limit the amount of code that must be verified to ensure security. It must be both possible and *convenient* to run code with reduced privilege.
3. Developers expect a general programming model, including support for distributed programs and concurrent threads with shared variables.
4. Developers need to do all of the above in the context of a familiar programming language.

This paper introduces *Aeolus*, a new DIFC platform developed to satisfy the above requirements. Aeolus is designed for building distributed applications, such as a web service that stores many users' data on multiple servers, or a medical records system accessed from computers in doctors' offices. Aeolus addresses two data security issues: *confidentiality* and *integrity*. Confidentiality ensures that secrets cannot leak except through explicit privileged operations. Integrity ensures that data from untrusted sources is not trusted inadvertently.

Aeolus supports the first requirement by providing a new security model with simple rules based on *principals* and *tags*. Principals represent entities with security interests, and tags allow principals to categorize information. Both are familiar real-world concepts that developers are accustomed to reasoning about. This model allows for structured, fine-grained delegation of authority, through an *authority state*. The authority state allows policies to be specified declaratively and can be used to enforce mandatory *policy constraints* such as separation of duties. Moreover, Aeolus readily supports *revocation*.

Aeolus meets the second requirement by providing new abstractions that support the principle of least privilege [18]. Support for this principle is important for im-

plementing secure applications, but if the mechanisms for limiting privilege are not convenient they will rarely be used in practice – as anyone who has attempted privilege separation on Unix knows all too well. Aeolus's runtime environment provides programmers with the ability to invoke functions with reduced authority, and provides *authority closures*, which allow authority to be delegated to particular programs without fear that that trust can be misused to run other code.

To support the third requirement, Aeolus provides a number of additional abstractions. Aeolus allows programs to run concurrent threads with different levels of contamination. It uses a memory-safe language to isolate threads from each other, while providing a low-overhead *secure shared state* mechanism that allows for efficient sharing while still enforcing information flow restrictions. Aeolus supports distributed programs with a secure RPC mechanism, and provides *boxes*, which allow confidential information to be communicated without contaminating intermediaries that do not observe the information.

Aeolus is a dynamic DIFC system implemented in a set of runtime libraries. Thus, it is OS-independent, and allows programmers to write code in a familiar and conventional programming language, thereby addressing the final requirement. The Aeolus runtime environment runs on all nodes in a distributed system, and allows communication between them while enforcing information flow restrictions. This paper describes our implementation of Aeolus for Java, and shows that it has low performance cost. We have also ported parts of Aeolus to C# and PHP.

Aeolus differs from but is inspired by both streams of current DIFC research: programming languages that enforce static restrictions on information flow [14, 15, 17], and operating systems that track information flow dynamically [6, 9, 23, 24]. Aeolus has more in common with the operating system work, as it tracks information flow dynamically at the granularity of threads rather than requiring programmers to specify restrictions at the level of individual variables. However, because it operates at the language runtime level rather than the OS level, it can provide higher level abstractions for writing secure programs, such as threads with secure shared state. Whereas existing DIFC operating systems manage authority with capabilities, Aeolus uses the authority state. This readily supports revocation and policy constraints, which are difficult to achieve in capability systems.

2 Aeolus Architecture

Aeolus provides information flow control for a distributed computing environment; the architecture is shown in Figure 1. The system consists of many nodes, each of which runs the Aeolus platform and is trusted to enforce Aeolus's information flow rules. Accordingly, only trusted nodes are allowed to enter the system.

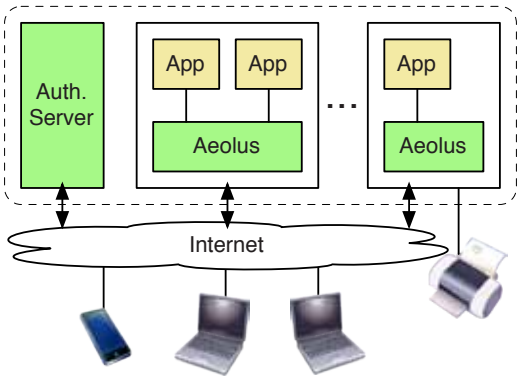


Figure 1: System Architecture

Aeolus tracks information flow within each system node and between nodes. It allows sensitive information to flow between system nodes, but the messages are encrypted and authenticated so that secrecy and integrity are protected. Aeolus restricts communication with nodes outside the system and to I/O devices, all of which are considered to be untrusted. Information can flow out of the system only if it is uncontaminated, and information arriving from the outside is marked as having no integrity.

Aeolus vests authority in principals, which it represents by principal IDs; each thread in Aeolus runs for some principal. Similarly to other dynamic information flow control systems, Aeolus tracks information used by programs as they execute, and determines whether programs have the authority to perform security-sensitive operations, such as declassification. This requires a way to track the authority of principals. Aeolus does this through *authority state*; this is shown as residing at a single authority server (AS), but a distributed implementation is also possible. The AS also tracks system membership; only nodes registered at the AS are in a deployment.

Threat Model. Aeolus is aimed at preventing errors and malicious behavior from undermining information security. The goal is to allow applications to minimize the amount of their code that needs to be trusted, following the principle of least privilege. Like much prior work on information flow control, we do not attempt to address covert-channel and side-channel attacks by malicious software running in the system, or by users who might transcribe data from the screen. However, we have been careful to ensure that our mechanisms do not introduce additional covert channels.

The trusted computing base (TCB) for our system consists of our platform code, together with the lower layers on which it runs: the OS and hardware. We also require a secure authentication service. In our prototype, we implement our platform atop the Java Virtual Machine (JVM), and therefore the TCB includes this as well.

Because the trusted computing base includes a com-

Principals.

- `createPrincipal()` $\rightarrow P$. Returns a new principal; the creating process's principal acts for P.
- `actsFor`(P_1, P_2). Adds an acts-for link from P_2 to P_1. The process must act for P_1, and the link must not create a cycle.
- `revokeActsFor`(P_1, P_2). Removes the acts-for link from P_2 to P_1 (if one exists); the process must act for P_1.

Tags.

- `makeTag()` $\rightarrow t$. Returns a new tag; the process's principal is authoritative for t.
- `makeSubtag`(t_1) $\rightarrow t_2$. Returns a subtag of a top-level tag t_1. The process must be authoritative for t_1, and becomes authoritative for t_2.
- `grant`(t, P_1, P_2). Adds a delegation link for t from P_1 to P_2. P_1 must be authoritative for t, the process must act for P_1, and the link must not create a cycle.
- `revokeGrant`(t, P_1, P_2). Removes the delegation link for t from P_1 to P_2 (if one exists); the process must act for P_1.

Labels.

- `addSecrecy`(t). Adds t to the secrecy label.
- `declassify`(t). Removes t from the secrecy label. The process must be authoritative for t.
- `removeIntegrity`(t). Removes t from the integrity label.
- `endorse`(t). Adds t to the integrity label. The process must be authoritative for t.

Figure 2: Some operations for principals, tags, and labels.

modity operating system and the JVM, Aeolus has a larger TCB than many DIFC operating systems. Our focus in this work has not been on reducing TCB complexity but on identifying the right abstractions for developers to build secure applications; providing a minimal-TCB implementation of the same abstractions is an interesting direction for future work. In addition, we believe supporting existing operating systems and languages is important for the system to be adopted.

3 Information Flow Model

This section describes the basic concepts and rules of the Aeolus security model. Figure 2 shows part of the API. The complete API is described in the Aeolus reference manual [1].

3.1 Principals, Tags, and Labels

The Aeolus model is based on three key concepts: principals, tags, and labels. *Principals* represent entities with security interests, such as individuals or companies. *Tags* provide a way for principals to categorize their information. For example, a user Bob might define three tags, for his public, financial, and medical information.

Labels are sets of tags and are used to control information flow. Each data object (such as a file) and each thread has two labels: a secrecy label, L_S, which reflects confidentiality of information, and an integrity label, L_I, which reflects the integrity of information. The labels of data objects are immutable: they are assigned when the object is created and cannot be changed. Thread labels are mutable: as a thread executes, its labels can change to reflect the secrecy and integrity of the information the thread has observed, subject to the rules defined below.

Aeolus maintains *security state* for each thread, consisting of its two labels and its associated principal.

3.2 Information Flow Rules

Information flow from a source S to a destination D is allowed only if two rules about their labels are satisfied:

- Secrecy Rule: $S.L_S \subseteq D.L_S$
- Integrity Rule: $S.L_I \supseteq D.L_I$

The secrecy rule ensures that confidentiality is maintained as data propagates, while the integrity rule keeps track of influences of low-integrity entities. These rules are an instantiation of the conventional lattice-based rules [4].

A thread can manipulate its labels by adding and removing tags. Adding a tag to a secrecy label and removing a tag from an integrity label are safe manipulations, since the thread only increases its contamination or reduces its integrity. However, the following *privileged* label manipulations are unsafe because they remove constraints on information flow:

- *Declassification*. Remove a tag from a secrecy label.
- *Endorsement*. Add a tag to an integrity label.

Therefore, a thread can perform privileged label manipulations only when its principal *has authority for* the tag being added or removed. Section 3.3 discusses authority

All label manipulations must be done *explicitly*. Aeolus differs in this respect from existing DIFC operating systems, which declassify automatically when a thread with authority for a tag reads an object with that tag in its label [6, 23]. Forcing programmers to be explicit prevents leaks due to unintended uses of authority.

3.3 Authority

Authority determines whether a thread can perform privileged label manipulations (declassification and endorsement). Authority starts with tag creation: when a thread creates a tag, its principal has authority for that tag. Subsequently, authority can be delegated either via *acts-for* relationships or via *grants*. Furthermore, previously delegated authority can be *revoked*.

When a new principal is created, the principal of the thread that creates it acts for it and thus has all authority of the new principal. Subsequently, a thread that acts for a principal can delegate that authority to other principals.

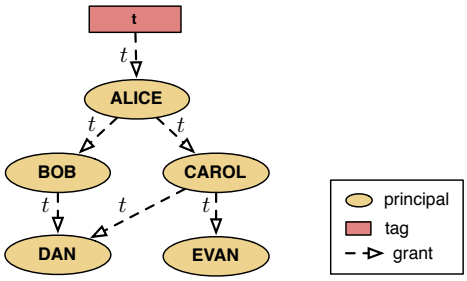


Figure 3: Delegation graph for a tag t

Information about acts-for relations is maintained in the *principal hierarchy*, which is a directed acyclic graph.

The principal hierarchy is useful to capture authority relationships for organizations (groups), and also allows individuals to use different principals for different purposes (roles). A special principal, P_{PUBLIC}, acts for no principals, and can be used to run a computation with no authority. All principals act for P_{PUBLIC}.

Delegation through the principal hierarchy is a blunt instrument, because when one principal acts for another it has *all* the authority of that principal. Grants provide a safer, more controlled, delegation of authority: a principal can grant its authority for a particular tag to another principal. Just as tags allow users to categorize information and provide separate controls for different categories, grants allows users to control authority over those categories.

The delegation hierarchy for a tag is also a directed acyclic graph, as shown in Figure 3; each tag has its own graph. Here, tag t was created by principal ALICE, ALICE has delegated authority for t to both BOB and CAROL, BOB has delegated his authority to DAN, and CAROL has delegated authority to DAN and EVAN. The principals shown in the figure all have authority for t, and so do any principals that act for them either directly or transitively.

Revocation. Revocation of authority is an important concern in real systems. Aeolus allows both acts-for relationships and grants to be revoked. Revocation removes a particular link from the principal hierarchy or delegation graph, and that revocation is transitive. For example, if ALICE revokes her delegation to CAROL in Figure 3, CAROL and EVAN lose authority for t but DAN does not.

Of course, performing a revocation requires authority. If P_2 acts for P_1, that link can be revoked only by a thread that acts for P_1. Similarly, a grant can be revoked only by a thread that acts for the grantor.

3.4 Compound Tags

Applications frequently have sets of tags that are closely related. Capturing such a relationship makes it easier to understand the authority structure of the application and more efficient to run the application.

Aeolus allows developers to express such relationships with *compound tags*. This mechanism allows tags to be grouped statically, as they are created. For example, the medical records system we describe in Section 5.1 has a tag for each patient's data, each a subtag of the ALL-PATIENT-DATA compound tag. Compound tags simplify delegations and label manipulations. Authority for the entire group of tags can be granted by delegating authority for the compound tag. Compound tags reduce label size substantially, since the label only needs to contain the top-level tag, and make declassification inexpensive as only the top level tag must be removed.

3.5 Manipulating Authority State

Aeolus maintains *authority state*, which consists of the principal hierarchy, tags, and their delegations. Applications can modify the authority state by creating principals and tags, or by delegating or revoking authority. These modifications create opportunities for covert channels through the authority state. For example, a malicious application could leak secret information by granting authority for certain tags to a co-conspirator based on the contents of the secret; the co-conspirator then learns about the secret by observing which tags it was granted authority for.

We avoid these covert channels by permitting only threads with null secrecy labels to modify the authority state (i.e., the authority state itself is an object with a null secrecy label). We believe this is a reasonable restriction, as modifications to the principal hierarchy are rare and typically do not occur during normal computation. For instance, authority state is modified when a new user is created but not when that user performs operations. Alternate approaches that allow the principal hierarchy to be modified by contaminated threads are possible, at the cost of increased complexity [19, 20].

3.6 Policy Constraints

An important concern in systems that support dynamic security policies is ensuring that the policies themselves are correct. An example of a correctness requirement is separation of duties between doctors, who can view their patients' sensitive medical data, and administrative assistants, who can view billing and insurance records: no principal ought to be authoritative for both roles.

Such invariants can be enforced by stating *policy constraints*, which are predicates that the authority state must satisfy. Aeolus prevents modifications to the authority state that violate a policy constraint. To define a constraint, a principal must have authority for every principal and tag the constraint covers. The ability to define constraints over an explicit principal hierarchy is an advantage of Aeolus's authority model over capability-based systems.

4 Programming Model

This section describes the programming abstractions Aeolus provides, and explains how they support practical DIFC and the principle of least privilege.

4.1 Threads and Virtual Nodes

Aeolus applications consist of multiple threads, each with its own security state (principal, secrecy label, and integrity label). Aeolus threads can share memory, but their accesses must obey the information flow restrictions. Each thread's memory is private; other threads cannot access it. Threads can share objects only through Aeolus's shared state mechanisms, described in Section 4.2, which enforce the information flow restrictions.

Distributed applications can run on multiple physical machines, and multiple applications can be run on the same physical machine. To support this, each thread is part of a *virtual node*. Each virtual node runs on a single physical node and may contain many threads. Typically, an application will run one virtual node on each physical node it uses. Virtual nodes are used to isolate applications: as we describe below, only threads in the same virtual node can share memory. Threads can also communicate with other virtual nodes via RPC as discussed in Section 4.4, and through Aeolus's distributed file system.

Importantly, shared state, RPCs, and the Aeolus file system are the *only* mechanisms by which Aeolus allows threads to communicate with each other. All these mechanisms check the threads' secrecy and integrity labels. Thus, information cannot leak from one thread to another if it is not permitted by the information flow rules of Section 3.2.

4.2 Shared Objects and Boxes

Aeolus allows threads to securely share state, while enforcing information flow restrictions. The mechanisms ensure that each thread's labels accurately reflect information communicated through shared state.

Aeolus uses two rules to enforce secure sharing. First, each object in shared state has labels, and a thread can only read or modify the object if permitted by the standard flow control rules. Second, shared objects are *encapsulated*: threads cannot obtain pointers to the interior of shared objects, which prevents the threads from bypassing the label checks. Similarly, shared objects cannot contain pointers to the local memory of any thread. To ensure proper encapsulation, Aeolus performs deep copies of arguments and results of calls on shared objects: it recursively follows pointers and copies objects, except for pointers to other encapsulated shared objects.

Users can define shared objects with arbitrary methods; the Aeolus platform adds runtime label checks and copies arguments to ensure encapsulation. Aeolus conservatively assumes that each method of a shared object both reads and writes the object (a flow out and a flow in, respectively). Therefore, calls are allowed only if the labels of the thread match those of the object *exactly*. In addition, the thread's label cannot be changed while executing a shared object method.

Aeolus provides three kinds of built-in shared objects with less restrictive label rules. *Shared queues* provide a form of IPC and *shared locks* provide IFC-aware synchronization among threads. *Boxes* are in-memory containers whose labels reflect the contamination of the information inside the box. A thread can copy data into the box or copy data from the box if its labels and the box's labels allow the flow.

Boxes provide special semantics for RPCs (see Section 4.4): contaminated information can be sent inside a box and the recipient becomes contaminated only when it opens the box to retrieve the content. For example, a web server can receive a box containing a password from a user and pass it to an authentication service without looking at the password itself. Similar control could be achieved by using the file system, by having the caller send the pathname of a file containing the tagged information, but using boxes is simpler and more efficient.

Every virtual node has a *root* object that its threads can use to access shared state. This object has null labels and is typically used to locate other shared objects. A shared object is inaccessible when it cannot be reached from the root or from any thread. Storage for inaccessible shared objects is collected automatically.

4.3 Principle of Least Privilege

The principle of least privilege is essential for building secure applications, because it prevents bugs from becoming critical security failures. Aeolus must make it possible to ensure that each part of the application runs with only the authority it needs. More than that, it must make it *convenient* and *efficient* to do so, lest this principle fall by the wayside in practice. Temporarily dropping authority and regaining it should be as easy as making a function call.

Aeolus supports the principal of least privilege using two mechanisms: *reduced authority calls*, which allow a thread to drop privilege, and *authority closures*, which execute code with previously bound-in authority.

Reduced authority calls are straightforward: the caller specifies a function to run, and the principal to run it with. The caller must act for that principal (i.e. it must reduce authority, not raise it). We expect applications to use these calls frequently to drop privilege they do not need. Aeolus also provides reduced authority forks, which start new threads. During a fork, the arguments to the call must be copied into the memory of the new thread, so that it does not share memory with the old one.

An *authority closure* is an object that is bound to a

principal. The principal is specified when creating the closure; the thread that creates the closure must act for that principal. Thereafter, any thread can call methods of the closure. Calls start running with the labels of the caller, but the authority (principal) of the closure. On return, the caller's labels are merged into the thread's labels – a union for the secrecy labels and an intersection for the integrity labels. This allows the closure to use its authority to remove contamination added during its own execution, but it cannot remove contamination its caller already had.

Programmers create new authority closures by defining subclasses of AeolusClosure. Closure objects are vested with authority when they are instantiated: the AeolusClosure constructor is passed a principal, which the caller must act for. The Aeolus runtime treats closure objects specially: whenever one of their methods is invoked, the system switches the thread's principal to the closure's for the duration of the call and performs the label manipulations described above. An example of an authority closure is shown in Section 5.3.

4.4 RPCs, External I/O, and Files

Aeolus applications use RPCs to communicate between virtual nodes. An application makes a closure available for RPC by binding it to a name. When a remote thread invokes the RPC, a new thread in the closure's virtual node executes the call with the authority of the closure and the labels of the caller, like calling a closure locally.

A thread must have a null secrecy label to bind a closure to an RPC name since otherwise the existence of an RPC with a particular name could be a covert channel. This restriction is not problematic because applications typically register RPCs when they start.

Clients outside the system can send requests to Aeolus nodes using the RPC protocol or by using sockets. Data received from outside the system is given a null integrity label since we cannot vouch for its validity. Replies can only be sent outside the system if the sender's secrecy label is null, since we cannot guarantee confidentiality of data sent to the outside. Furthermore, boxes cannot be sent externally because external nodes are not trusted with the contents of the box.

Finally, Aeolus provides a network file system that enforces DIFC. Files have immutable labels and access is allowed only if the label constraints are satisfied; the rules are similar to those in HiStar [23].

5 Evaluation: Expressive Power

This section evaluates Aeolus with respect to the goals established in the introduction: understandable and flexible authority structure, support for the principle of least privilege, and support for distributed and concurrent programs. We do this by examining three examples, each representing a class of similar applications:

- The medical information system (Section 5.1) represents systems in which there are complex security relationships between individuals in an organization.
- Bob and the tax preparer (Section 5.2) represents distributed computations with two parties with different security interests, and demonstrates the use of boxes.
- The online personal finance service (Section 5.3) represents computations with many parties with separate security interests. It illustrates the use of shared state, and of reduced authority calls and authority closures to support the principle of least privilege.

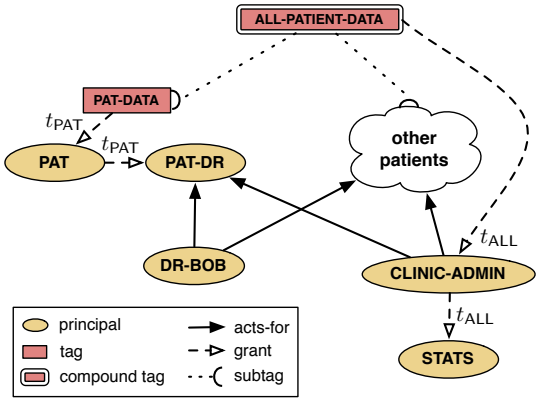


Figure 4: A portion of the authority state for a medical clinic. For each patient, there is a principal for the patient (PAT), a principal representing the patient's doctor (PAT-DR), and a tag (PAT-DATA) applied to the patient's records.

5.1 The Medical Clinic

Ensuring confidentiality and integrity of medical records is a major problem for the health care industry [11]. This example illustrates the authority structure of such a system, requiring delegation, revocation, and compound tags.

Figure 4 shows part of the authority state for a medical clinic. It focuses on a particular patient, Pat. Pat is represented by principal PAT, and has a tag PAT-DATA that is applied to all records containing Pat's protected health information. Accordingly, only threads running as principals with authority for PAT-DATA can declassify and expose Pat's information – for example, to display it on the screen or send a prescription to the pharmacy.

Only doctors caring for Pat should have authority for Pat's information. This constraint is achieved by having Pat grant authority for the PAT-DATA tag to a PAT-DR principal. Then, one or more doctors (in Figure 4, just DR-BOB) can act for the PAT-DR role. Importantly, these delegations can be changed. If Pat gets a new doctor, the CLINIC-ADMIN, who also acts for PAT-DR, can allow the new doctor to act for PAT-DR. If Pat changes doctors, the CLINIC-ADMIN can

revoke the acts-for link between PAT-DR and DR-BOB, thus ensuring that DR-BOB no longer has authority for Pat's data.

In addition to providing doctors access to patient records, the medical clinic also runs a statistical analysis package periodically. This code runs as an authority closure with principal STATS, which is delegated authority for all patient tags; the code is trusted to obfuscate the result. To perform the delegation, the medical clinic uses a compound tag, ALL-PATIENT-DATA; PAT-DATA and the other patients' tags are subtags of this compound tag. Using the compound tag makes the label small (it contains just the compound tag) and allows efficient declassification (only one check is required), even though there may be thousands of patients. The delegation to STATS also does not need to be changed when new patients are added.

Discussion. The medical clinic application benefits from Aeolus's authority model. It is natural to describe the relationship between Pat, Pat's doctor, and Pat's data using principals and tags. By examining the authority structure, one can easily answer the question of who can declassify Pat's medical records. Further, constraints can be enforced, such as requiring that only doctors can act for a doctor role. In existing DIFC operating systems, authority is managed by capabilities, so these policies would require additional application-specific trusted code to control access to the appropriate declassification capability.

This application also benefits from support for revocation: removing DR-BOB's access is simply a matter of revoking a particular delegation. Once this delegation is revoked, threads running on behalf of DR-BOB, or anyone to whom he might have delegated authority, can no longer declassify Pat's data. In systems that use capabilities, revocation is more challenging, because capabilities cannot typically be revoked: threads may already be running with the revoked authority, and most DIFC operating systems also allow capabilities to be stored persistently on disk. Modern capability systems address this problem by building higher-level abstractions atop capabilities [13]; Aeolus provides these abstractions directly.

5.2 Bob and the Tax Preparer

This example, from [15], illustrates the use of distributed computation and boxes. Here, Bob is a client of an online tax preparation service. He submits his financial information to the service, along with billing information that is used to charge Bob for the service. The tax preparer then uses a proprietary database to produce Bob's tax form. There are two secrecy goals: Bob's financial information and tax form are confidential and only Bob should be able to see them; and the tax preparer's database is confidential and should not be disclosed to Bob or anyone else.

In our implementation, Bob uses a client that communicates with the tax preparer via RPC; both run on Aeolus nodes. The RPC is handled by a thread with authority for the TAX-PREP tag. Bob's client places his information in a box with secrecy label {BOB} and sends the box in an RPC to the tax preparer, along with Bob's untagged billing information. The thread in the tax preparer's virtual node that executes the RPC records Bob's billing information. Then it adds tags BOB and TAX-PREP to its secrecy label, allowing it to open the box containing Bob's financial information and read the proprietary tax database. Next, it computes Bob's tax form, uses its authority to remove the TAX-PREP tag, and returns the form to Bob.

Discussion. This example benefits from Aeolus's support for distributed computation. In particular, Bob can send an RPC with arguments that contain labeled data, and rely on the tax preparer's Aeolus runtime to ensure secrecy. Most prior DIFC systems cannot communicate sensitive information over the network. The notable exception is DStar [24], but it forces applications to periodically refresh authority, which is inconvenient.

The use of boxes is essential in this example. The tax preparer does not have authority for BOB. If it became contaminated immediately upon receiving the RPC, it could not record the billing data in a file without BOB in the label. Boxes allow the thread to avoid becoming contaminated until it actually reads Bob's sensitive information.

5.3 Financial Management Service

Our final example, which we use as a benchmark in Section 7.1, is an online personal finance management service inspired by Mint.com. Users provide the system with online banking credentials for their bank accounts, and the service aggregates their transaction histories, computes statistics, and produces a report. The application uses files and shared state to store information securely, and uses authority closures and reduced authority calls to run code with minimal authority. Users and banks are external entities in this system, and do not run the Aeolus platform; all communication with these entities must thus be done with null labels.

There are several clear security requirements. A user's financial data must not be exposed to other users or third-party banks. A user's online banking credentials (username and password) are even more sensitive: they should not be used for any purpose other than to log in to the corresponding bank. They should not even be revealed to the user, in case the user's access to the site is compromised.

To capture these constraints, each user has a separate tag, e.g., ALICE-DATA and a principal, e.g., ALICE, that is authoritative for this tag. These tags are subtags of the ALL-USER-DATA compound tag. There is also a principal for each bank, and an associated tag that is used to protect users' credentials for that bank.

A user's financial data is stored in a file with the appropriate USER-DATA tag in its secrecy label. Each of a user's bank credentials is stored in its own file, with a secrecy label containing both the user's tag and the appropriate BANK tag; this way even the user can't expose the credentials.

Because a user's session might consist of many requests, the service also caches information for active users in a shared hash table that maps user IDs to session state for that user. Within a session-state object are boxes containing bank credentials for each of that user's banks. The labels here mirror those of the files: the secrecy label of the session-state object contains the USER-DATA tag, while each box has both the USER-DATA and the BANK tags in its secrecy label.

When Alice requests to display her recent transactions, the request is received (over a secure channel) by a thread that is authoritative for all users. After validating the user's identity, it makes a reduced authority call, dropping its authority to the ALICE principal.

The thread then locates the session-state object containing her information (assuming it is in the cache). Reading Alice's session-state object requires adding the ALICE-DATA tag to the thread's secrecy label. If a bug in the code caused the thread to read a *different* user's information, it will not be able to remove the corresponding tag and thus cannot leak it outside the system.

The thread then calls an authority closure for each of Alice's banks. This closure takes the box containing Alice's credentials for that bank, and obtains and returns a list of Alice's transactions from the bank; a sketch of the closure code is shown in Figure 5. The closure runs with authority for both its BANK tag and the ALL-USER-DATA tag. It adds the BANK tag to the thread's secrecy label, opens the box to obtain the credentials, uses its authority to declassify, and contacts the remote bank. When the closure returns Alice's transactions, its label is merged with that of its caller, so the user's tag, e.g., ALICE-DATA, is restored to the label. Therefore, we need not be concerned that Alice's data will be exposed to some other user.

After obtaining information from all banks, the thread computes the result. Since this involves invoking untrusted code – for example, an OFX parser or a graph-drawing library – it does so using a reduced authority call to P_{PUBLIC}. Because the thread's label contains the ALICE-DATA tag, the called code can process her financial data but cannot expose it over the network or store it in an unlabeled file. When the reduced authority call returns, the thread removes the ALICE-DATA tag from its secrecy label and returns the result to Alice.

Discussion. Aeolus' reduced authority calls and authority closures make it convenient to run application components with minimal authority, so only a small amount of application code needs to be trusted for security. For example, to ensure the secrecy of a user's bank password, we need only trust the bank closure, as only it runs with the authority to remove the bank's tag. In previous DIFC operating systems, dropping and regaining authority requires use of a different process, which is more difficult to program and imposes higher overhead – making it likely that developers will not bother to do so.

```
public class BankClosure extends AeolusClosure {
    private BankInfo bank;
    private Tag bankTag; // the tag for this bank

    public BankClosure(PID bankPID, Tag t, BankInfo b) {
      super(bankPID); // binds the closure to bankPID
      bankTag = t;
      bank = b;
    }

    // this function runs with bankPID authority
    public Transactions getTxns(Box<Credentials> u) {
        addSecrecy(this.bankTag);
        Credentials c = u.get()
        declassify(this.bankTag);
        declassify(c.userTag);
        Transactions t = download(bank.url,
                                  c.username, c.password);
        return t;
    }
}
```

Figure 5: Example of an authority closure

The application also benefits from shared state since it can cache data in memory rather than resorting to files. Aeolus is the first DIFC system to support secure shared state with labeled application-level objects. An application running on an existing DIFC operating systems could implement a "shared state manager" process that other processes communicate with by IPC, but this is inconvenient, expensive, and risky: that process needs to run with complete authority because it is contaminated by the content of all shared objects.

6 Implementation

This section describes some highlights of our Java-based implementation of Aeolus. Further details, including the complete interface, are available in the Aeolus reference manual [1]. We also have implementations for C# and PHP, but do not describe these here.

Figure 6 shows the structure of the implementation at an Aeolus node. Aeolus runs on top of the JVM within a single OS process, and all Aeolus applications on the node run within the same JVM. The Aeolus runtime runs a special thread (the authority state client) that manages an authority state cache and handles interactions with the authority state server (the AS) on behalf of all threads at the node. Application code accesses Aeolus features via a set of libraries. The AS runs on a separate machine.

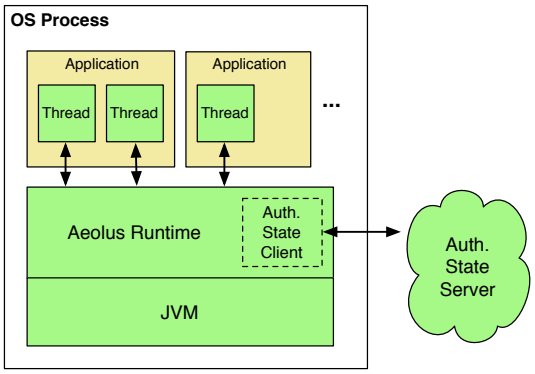


Figure 6: Structure of an Aeolus node

6.1 Threads and Isolation

All Aeolus threads on a node, even those belonging to different virtual nodes, run within the same JVM. All threads and shared state objects share a common heap.

Running all threads in the same JVM provides good performance: for example, it allows us to support fast shared state communication between threads. However, each Aeolus thread must appear to have its own isolated memory, and label checks must be interposed on accesses to shared state and boxes. Aeolus achieves this isolation goal by relying on the memory safety of the JVM: each thread can access only the parts of the heap accessible from its stack. Aeolus ensures that only one thread can have a pointer to any object, except for encapsulated shared objects.

Implementing Copy-Based Isolation. To ensure that only one thread can have a pointer to a non-shared object, objects must be copied before they can be passed to a new thread. Similarly, arguments to and results from shared object methods must be copied to prevent threads from obtaining pointers to the contents of encapsulated shared objects. These deep copies can be expensive; prior work rejected copying entirely, citing prohibitive costs [12]. Aeolus uses a highly optimized, low-level cloning library to reduce the cost of copying.

An important optimization is to avoid copying immutable state, which can be safely shared. Aeolus recursively analyzes classes at load time to determine whether instances could contain mutable (non-final) fields or can refer to any mutable objects via fields, superclasses, and inner and outer classes. For the purposes of this analysis, shared state objects are considered immutable; as we describe below, they are instrumented to perform runtime label checks, so references to them can be shared.

Programmers can use the *AeolusSafe* marker interface to indicate that a type is immutable. If an AeolusSafe type contains or refers to mutable state, a load-time error occurs. Aeolus can safely assume that all subtypes of an AeolusSafe type are immutable. If a type is AeolusSafe (or if it is immutable and final) then the instrumentation to copy values of that type is omitted entirely.

Restrictions. Aeolus requires application code to use a "safe" subset of Java, because certain Java features could be used to circumvent isolation and the information flow rules. Aeolus applications may not use:

- reflection and native code, which can bypass the Java type system
- static variables, which could allow threads to share state in ways that violate the information flow rules[1]
- direct access to underlying Java APIs for I/O and threading; applications must use the Aeolus API

The restrictions are enforced through a combination of load-time bytecode verification implemented with a custom class loader, and runtime checks implemented using Java's SecurityManager framework. Our approach to isolation is similar to that used by other systems that run multiple applications in a single JVM [3, 12].

6.2 Shared Objects and Closures

The Aeolus class loader uses bytecode rewriting to make shared objects and closures safe, convenient, and efficient. Shared state objects and closures can provide any number of public methods; the Aeolus class loader adds instrumentation to them to enforce the information flow rules. Closure methods are instrumented to save the caller's labels and merge them into the thread labels on return. Shared state objects are instrumented to perform the necessary label checks, and also to copy arguments and return values of method and constructor calls (including any thrown exception objects) if necessary to enforce isolation.

Closure objects are required to be immutable (subtypes of AeolusSafe). This prevents closures from keeping state between calls and potentially leaking information to differently labeled callers.

6.3 Management of Authority State

Authority state is stored in a transactional database at a special *authority server* (AS). Each node runs an *authority state client*, which sends all authority updates to the AS. Updates are infrequent, and thus result in minimal overhead, but queries over the authority state are common. The authority state client maintains a local cache of authority state to reduce query latency and AS load.

When there is a miss in the authority cache, Aeolus fetches a *block* of related information. Since the notion of what is "related" depends on the application (e.g., if we need information about a patient in a medical system,

[1] We plan to remove this restriction in the future by providing private versions of static variables for each thread. The technique is similar to that proposed for Java's as-yet-unavailable isolation API [3].

what other information is useful to know?), we provide a way for applications to organize the state into *blocks*.

The cache needs to be managed properly to ensure that it contains correct and timely information. When a block is fetched, the cache also receives and applies all updates that have occurred since its last communication with the AS. This ensures that each query runs on a consistent snapshot of the AS state. In addition, we provide causality by propagating information about the most recent AS update at the sender in its messages; at the receiver, we ensure that the next use of the cache reflects this update or a later state. Finally we provide timeliness, which is especially important for revocation. A cache stops processing queries until it can communicate with the AS if its update information is older than δ seconds; δ is a system parameter and might be on the order of 30 seconds.

The Query Cache. Even if all necessary blocks are present in the cache, determining whether a principal has authority for a certain tag might be computationally expensive if the application has a complex authority state. To avoid this overhead, we also maintain a *query cache*, which stores the results of recent queries. The query cache contains a set of pairs; each indicating that a particular principal has authority for a specific tag, or acts for a specific other principal.

When a client receives updates from the AS, it removes all query cache entries computed from information in blocks that have changed. This approach reduces the metadata needed for each query-cache entry (we store information about blocks rather than specific delegation links) but can lead to unnecessary evictions.

7 Evaluation: Performance

This section demonstrates that a large application running on Aeolus performs about as well as an application running on pure Java that does an equivalent amount of work. We first examine the end-to-end performance of an implementation of the financial management service from Section 5.3, and show that the overhead of adding information flow control with Aeolus is minuscule. To gain further insight into this result, we then explore the sources of overhead, via microbenchmarks.

A key contributor to Aeolus's low overhead is that it tracks information flows at the level of threads, relying on memory safety for isolation. Therefore, protection boundary crossings are much cheaper than in information flow systems that rely on OS processes for isolation [6, 9, 23]. Furthermore, execution is cheaper than in systems that do finer-grained tracking [16, 22] because Aeolus interposes only on communication and not on individual memory accesses. Some of our optimizations, particularly authority query caching, fast copies, and immutability analysis, also contribute substantially to our performance results.

These experiments use our Java implementation of Aeolus on a 2.50 GHz Core 2 Quad system with 4 GB of RAM, running Ubuntu 11.04 with Linux kernel 2.6.38-10. Our Aeolus libraries ran atop the OpenJDK 1.7.0_1 JVM.

7.1 End-to-End Performance

The evaluation of the financial management service prototype is interesting because it tests the performance of a wide range of Aeolus features: the code does label manipulations, makes use of shared state, and uses both authority closures and reduced authority calls. The prototype operates as a single virtual node that receives requests from users, fetches their user information, and retrieves financial data using bank closures. In the experiment, each user has three banks, and we use ten threads to handle user requests (i.e., we can run ten requests at a time). Each bank closure establishes an SSL connection to a user's bank, downloads the user's bank statement in OFX format, and returns the result; lacking real banks to test with, we simulate the network delay by sleeping for 100 ms. When all bank information has arrived, the thread generates a graph of spending habits, using a reduced authority call.

We compared the average request processing time for the implementation of this benchmark to one that does not use Aeolus or DIFC. The Aeolus version required 323.9 ms processing per request, 0.15% greater than the 323.5 ms processing time for the native version. There was a corresponding decrease in throughput from 17.97 req/s to 17.61 req/s. We observed similar overhead when varying the amount of time taken to process financial information retrieved from each bank.

7.2 Microbenchmarks

Applications running on Aeolus have low overhead, as shown above, because they invoke security operations relatively infrequently and mostly do real work. Nevertheless, it is worthwhile to examine the sources of overhead.

Shared State and Copying. Aeolus must check the caller's labels on each call to a method of a shared state object, but this overhead is negligible because labels are small (typically they contain one tag), making shared state a viable option for communication between threads. A trivial call with an empty label and no arguments or return value costs 8.9 ns.

The cost of copying arguments and return values of shared state methods is more significant, however. As discussed in Section 6.1, Aeolus avoids copying these objects if they are immutable; it takes 13 ns to make this determination by looking up the object's type in a hash table. If a formal parameter to a method is known to be immutable at load time (i.e., it is a primitive type, a

Operation	Time (ns)
Reduced authority call to P_{PUBLIC}	7.7
Reduced authority call to P_x	51
Closure call	83
Java method call	4.0

Figure 7: Time to perform a reduced authority call or invoke a closure. The cost of an ordinary Java method call is provided for comparison.

final and immutable class, or a shared or AeolusSafe object), then the copying instrumentation is omitted entirely, reducing the overhead for that argument to zero.

For objects that must be copied, Aeolus uses a fast, low-level mechanism that takes about 93 ns to duplicate an object, plus the time to recursively duplicate its fields. This offers significantly greater performance than the typical implementation of cloning, which serializes the object into a buffer and then deserializes the buffer; this naive approach takes 6.3 μs to copy an empty object, much of it spent on needless validity checks. Most complex objects contain at least some state that is immutable, e.g., Strings, so our optimization of avoiding copies of these fields is also significant.

We found that copying array elements was particularly slow (200 ns). Arrays of immutable elements (such as bytes, integers, or Strings) are common, so we made them a special case. Using `System.arraycopy` reduced the per-element copying cost to 1.5 ns for an array of immutable objects.

Closure and Reduced Authority Calls. The table in Figure 7 shows the overhead involved in performing reduced authority calls and closure calls when the process label is empty and there are no arguments or results. Calls to P_{PUBLIC} incur less overhead because they need not check authority state.

Forks. Aeolus can fork a new thread running with different authority and start running in the new thread in 12 μs. Tracking security state and enforcing isolation adds a small overhead relative to the 4.7 μs it takes to start a thread in Java. (Thread pools are used as an optimization in both cases.) Because Aeolus implements isolation using threads, these figures are much smaller than the cost to fork an OS process, the equivalent operation in a DIFC operating system. For comparison, creating a new OS-level process on the same machine requires 135 μs.

Authority Management. Aeolus adds overhead in the management of authority state. In particular, a thread can only perform a declassification or endorsement on a tag if its principal has authority for that tag. In applications with complex authority structure, checking this may require traversing a number of delegation links.

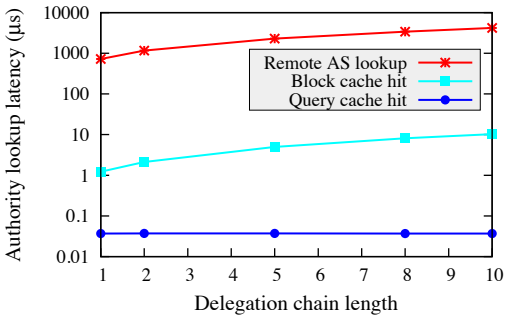


Figure 8: Authority lookup cost, by cache level used. Note the logarithmic y-axis.

Our use of a two-level cache mitigates this cost. Figure 8 shows the average request latency for an authority check, with varying delegation chain lengths. Depending on the length of the chain, it takes 0.7 to 4.2 ms to fetch the relevant state from the authority server to answer a query. If the state is already in the block cache, the answer can be computed locally in 1 to 10 μs. If the answer is in the query cache, it takes only 37 ns, regardless of the length of delegation chains. Here, the authority server is located on the same local network as the client. In a wider-area deployment, latency for uncached requests would be higher and the cache's benefit would be even greater.

File I/O. Aeolus must interpose on accesses to the file system to enforce information flow control. Because it must load and check the file's label, Aeolus imposes an additional cost of 400 μs on the first read to a particular file. Subsequent requests are cached.

8 Related Work

Aeolus builds on work on DIFC in programming languages and operating systems, fusing the concepts developed in these two areas into a new high-level model. It also provides new mechanisms not found in other systems, such as closures, boxes, and shared volatile state.

8.1 Programming Languages

Jif [14, 15] introduced the DIFC model, which formed the basis for many subsequent systems, including Aeolus. Jif embeds information flow policies in labels and makes labels part of the type system; programs that violate the information flow rules are rejected by the compiler. Jif also supports certain kinds of dynamic policies. Aeolus, in contrast, does all label checks at runtime and does not require type annotations.

Some language-based approaches [2, 19, 20] use the concept of a dynamic principal hierarchy to specify security policies, and support precise fine-grained declassification. Aeolus's principal hierarchy draws inspiration

from this work. However, tags allow us to group objects with the same security policy in a more convenient way than is possible in the type-system-based approaches. Our model also operates at a coarser granularity, allowing programmers to focus on the privileges required for different modules of an application, rather than the sensitivity of individual variables and objects.

Fabric [10] adds trust relationships to the principal hierarchy, supporting federated systems with mutually distrustful nodes. Aeolus assumes all nodes in a deployment are trusted, but could be extended to use this approach. Fabric nodes cache the transitive closure of all the acts-for links they know about; similarly, Aeolus's query cache stores edges in the transitive closure of the authority graph, but Aeolus doesn't precompute the entire transitive closure.

Another significant difference between Aeolus and previous language-based approaches is that Aeolus does not require applications to be written in a new language, nor does it require special compiler support.

8.2 Operating Systems

DIFC-based operating systems expose information flow controls to the applications via the operating system API. Asbestos [6] and HiStar [23] are new operating systems that provide DIFC properties using labels and tags. Asbestos tracks information flow at the level of processes exchanging unreliable messages; HiStar acts at the microkernel level of threads, memory segments, and gates (which are somewhat like our authority closures).

Aeolus borrows the notion of tags and labels from this recent work. However, labels in Asbestos and HiStar combine mechanisms for privacy, integrity, authentication, declassification privilege, and access control. Flume [9], like Aeolus, separates information flow labels from authority and access control to make labels easier to understand. Also like Aeolus, Flume runs in user mode on a standard OS. However, Aeolus differs from Flume because it uses an explicit principal hierarchy rather than capabilities.

Aeolus also exposes higher-level abstractions to the application, such as closures and boxes. Whereas Asbestos, HiStar, and Flume require separate processes for privilege separation, Aeolus's abstractions make defining trust boundaries within an application easier and more efficient.

Among these systems, only HiStar provides support for shared memory, by exposing low-level page table protection mechanisms. Aeolus provides more usable sharing at the level of objects applications use. Laminar [16] also aims to provide sharing of application objects, using a hybrid of OS and JVM mechanisms. Laminar tracks information flow at object granularity; however, because fine-grained dynamic tracking is expensive, it only tracks contamination within code blocks called *security regions*. It does not track contamination outside these regions, making it hard to enforce end-to-end security guarantees.

DStar [24] extends HiStar over the network and is the only DIFC system that provides support for revocation. It does this by requiring authorizations to be refreshed periodically. Aeolus uses a similar technique internally but provides a more convenient abstraction to users.

Aeolus applications enforce discretionary information flow policies through carefully controlled use of authority – a notion programmers are familiar with. Other systems have explored alternatives such as data-flow assertions [22] and special-purpose policy specification languages [5].

The Singularity operating system [7], while not an information flow system, supports language-based isolation like Aeolus. Singularity provides fast IPC via hand-off: threads can exchange shared objects, but at most one thread can have a reference to a given shared object at any time. In contrast, Aeolus shared objects can be accessed by many threads concurrently, and Aeolus uses copying to prevent direct references from crossing the boundary between threads and shared objects.

9 Conclusions and Future Work

Aeolus is a new distributed platform for developing and deploying secure applications using DIFC. It provides a new security model based on tags and a principal hierarchy, which allows fine-grained delegation and revocation of authority. Recording trust relationships in a principal hierarchy makes it possible to define policy constraints.

Aeolus provides abstractions to make writing secure applications easier. Authority closures and reduced authority calls support the principle of least privilege. In addition, Aeolus supports distributed and concurrent programs, providing boxes to limit contamination, IPC, and shared state to allow convenient yet safe use of shared volatile information.

We implemented Aeolus in Java, and used it to develop several applications. Aeolus's features made expressing the desired information flow constraints convenient. Our experiments show that Aeolus has good performance, and its approach for caching authority state is effective. An initial release of the system can be downloaded from `http://pmg.csail.mit.edu/aeolus/`.

We are investigating programming language extensions to make Aeolus even more convenient. For example, it would be useful to have a syntactic extension for running blocks of code with reduced authority, eliminating the need to make reduced authority calls to separate methods. Also, if it were possible to recognize that a method on a shared object did not perform any mutation operations, either directly or indirectly, we could allow user-defined shared objects with relaxed label restrictions; presently, Aeolus conservatively assumes that shared object meth-

ods both read and write the object.

Additionally, we are currently extending the system with support for secure audit trails, and we are developing an approach to integrate databases into the model in a flexible way.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Eddie Kohler, for their helpful feedback. This research was supported by the National Science Foundation under grant CNS-0834239 and by funding from Northrop Grumman Corporation.

References

[1] Aeolus reference manual. Available at http://pmg.csail.mit.edu/aeolus-ref.pdf.

[2] S. Chong, K. Vikram, and A. C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *USENIX Security 2007*, Boston, MA, Aug. 2007.

[3] G. Czajkowski. Application isolation in the Java virtual machine. In *OOPSLA '00*, Minneapolis, MN, Oct. 2000.

[4] D. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[5] P. Efstathopoulos and E. Kohler. Managable fine-grained information flow. In *EuroSys '08*, Glasgow, UK, Apr. 2008.

[6] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *SOSP '05*, Brighton, UK, Oct. 2005.

[7] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in singularity OS. In *EuroSys '06*, Leuven, Belgium, Apr. 2006.

[8] J. A. Hall and S. L. Liedtka. The Sarbanes-Oxley act: implications for large-scale IT outsourcing. *Commun. ACM*, 50(3):95–100, 2007.

[9] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *SOSP '07*, Stevenson, WA, Oct. 2007.

[10] J. Liu, M. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: A platform for secure distributed computation and storage. In *SOSP '09*, Big Sky, MT, Oct. 2009.

[11] R. T. Mercuri. The HIPAA-potamus in health care data security. *Commun. ACM*, 47(7):25–28, 2004.

[12] M. Migliavacca, I. Papagiannis, D. M. Eyers, B. Shand, J. Bacon, and P. Pietzuch. DEFCON: High-performance event processing with information security. In *USENIX '10*, Boston, MA, June 2010.

[13] M. S. Miller and J. S. Shapiro. Paradigm regained: Abstraction mechanisms for access control. In *ASIAN '03*, Mumbai, India, Dec. 2003.

[14] A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL 1999*, San Antonio, TX, Jan. 1999.

[15] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP '97*, Saint Malo, France, Oct. 1997.

[16] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: Practical fine-grained decentralized information flow control. In *PLDI '09*, Dublin, Ireland, June 2009.

[17] A. Sabelfeld and A. C. Myers. Language-based information flow security. In *IEEE JSAC*, volume 21, Jan. 2003.

[18] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. In *SOSP '73*, Yorktown Heights, NY, Oct. 1973.

[19] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic. Managing policy updates in security-typed languages. In *CSFW '06*.

[20] S. Tse and S. Zdancewic. Run-time principals in information-flow type systems. *ACM Trans. Program. Lang. Syst.*, 30(1):6, 2007.

[21] U.S. Department of Defense Computer Security Center. Trusted computer system evaluation criteria. DoD 5200.28-STD, Dec. 1985.

[22] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *SOSP '09*, Big Sky, MT, Oct. 2009.

[23] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI 2006*, Seattle, WA, Nov. 2006.

[24] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control. In *NSDI 2008*, San Francisco, CA, Apr. 2008.

TreeHouse: JavaScript sandboxes to help Web developers help themselves

Lon Ingram*† and Michael Walfish*
*The University of Texas at Austin †Waterfall Mobile

Abstract

Many Web applications (meaning sites that employ JavaScript) incorporate third-party code and, for reasons rooted in today's Web ecosystem, are vulnerable to bugs or malice in that code. Our goal is to give Web developers a mechanism that (a) contains included code, limiting (or eliminating) its influence as appropriate; and (b) is deployable today, or very shortly. While the goal of containment is far from new, the requirement of deployability leads us to a new design point, one that applies the OS ideas of sandboxing and virtualization to the JavaScript context. Our approach, called TreeHouse, sandboxes JavaScript code by repurposing a feature of current browsers (namely Web Workers). TreeHouse virtualizes the browser's API to the sandboxed code (allowing the code to run with few or no modifications) and gives the application author fine-grained control over that code. Our implementation and evaluation of TreeHouse show that its overhead is modest enough to handle performance-sensitive applications and that sandboxing existing code is not difficult.

1 Introduction

This paper is about TreeHouse, a system that allows Web applications to safely include—in the code delivered to the browser—third-party modules that are unaudited, untrusted, and unmodified. By *Web application*, we mean any Web site that includes JavaScript.

Many of today's Web applications are closely integrated with third-party code and in fact depend on the correctness of that code. For example, *frameworks* are now in wide use; these are JavaScript libraries that serve as application platforms by abstracting messy aspects of the browser's interface [13, 19, 30, 45, 63, 67]. As another example, sites selling advertising space today include scripts supplied by *ad networks*; these scripts are only supposed to display content from the ad networks, but they can hinder or harm the enclosing page (even if running in a frame, as we explain below and in Section 2.3). A third example is *widgets*: code supplied by an off-site service to invoke that very service (for example, [7, 56]). To perform its function, the widget needs read and write access to the enclosing page.

Adding to the helplessness of applications, the third-party code can change unilaterally. Applications often include frameworks by *hyperlinking elsewhere*: for low latency, some frameworks' code is hosted by Content Distribution Networks (CDNs).[1] Similarly, applications include ad scripts and widget scripts by hyperlinking to the ad network or widget implementer. All of these cases are analogous to a desktop application that dynamically links to a module running on someone else's computer!

Web applications, then, are taking the risk of adding large, opaque, third-party code to their trusted computing base [12]. Whether from malice or bugs, this code can compromise the privacy of data [8], the integrity of the enclosing page [26, 57], and the availability of the application [8, 52] (for instance, the script can make many HTTP requests, slowing the page's load time).

Given this situation, our high-level goal is a mechanism by which Web developers can contain and control included code, whether written by a third-party or the Web developers themselves. The question that we must answer is: what interface should this mechanism expose, and how should we implement it?

Of course, this question is not new (not even in the Web context); our point of departure is in adopting the following two requirements:

- *Make it work today (or failing today, shortly).* Previous projects are more tasteful than ours; indeed their principles have inspired this paper. But realizing those principles has required deployability compromises that we hope to avoid. Specifically, we wish to minimize (a) browser modification [10, 14, 29, 33, 37, 38, 43, 59, 62] or redesign [5, 9, 16, 24, 39, 47, 53, 60], as these require changes in the entire Web ecosystem; (b) development-time code changes [11, 20, 35, 41], as these often impose a performance cost and always require framework authors to rewrite their code; (c) runtime code changes [44, 46], as these either sacrifice the performance benefit of hosting framework code in a CDN, or else incur a large performance cost in the browser (Section 7 explains further); and (d) server configuration, such as domain names for each trust domain [28, 31, 32, 64], as this impairs deployability.
- *Allow controlled, configurable influence.* The mechanism should allow only the access and grant only the resources needed for the contained script to do its job: frameworks should be given access to all elements in the enclosing page, widgets should be given access only to the portions of the page they are concerned with, and ads should have no influence on the enclos-

[1] Unfortunately, application code cannot even compare the hyperlink's target to a known content hash, owing to the same-origin policy (§2.1).

ing page. We note that browsers' *iframe* mechanism fails this requirement since code in a frame can still leak data or consume scarce browser resources owned by the application (see Section 2.3 for more detail).

TreeHouse's high-level approach is to provide a *sandbox* in which a Web application can run guest JavaScript code. Sandboxing (or jailing) on hosts [21, 22, 36, 49, 51, 58] and in browsers [15, 61] is an elegant way to run legacy code inside a given context while giving that code little (or configurably limited) influence on that context. These works restrict the *machine code* and *system calls* that the sandboxed code executes. Our scenario, however, calls for configurable control over code that has been programmed to the *browser's JavaScript interface*. Thus, we borrow the top-level idea of sandboxing but need an interposition mechanism that understands JavaScript and the browser's API.

Unlike other work that provides such interposition [10, 14, 33, 37, 38, 43], TreeHouse requires no browser changes: it is implemented in JavaScript using browsers' current functionality. Specifically, it repurposes *Web Workers* (a feature in recent browsers in which a page can run a script in a separate thread) as containers to run *guest code*. It avoids modification in that code by *virtualizing* the principal interface to the browser (known as the Document Object Model or DOM; see Section 2.1).[2] TreeHouse exerts configurable control over guest code using an approach analogous to trap-and-emulate in the virtual machines context [4]: it interposes on privileged operations, permitting them as appropriate.

We have implemented and evaluated a prototype of TreeHouse. We ported a benchmark suite [1], a Tetris clone [50], and two frameworks [45, 67] to TreeHouse. Using existing code with TreeHouse requires modest effort. TreeHouse's relative overhead for DOM operations is high, but its absolute costs are tolerable (to human users). With a small amount of engineering work, our prototype could be made ready for actual production use.

TreeHouse has a number of limitations. First, its trusted computing base (TCB) includes the browser and thus is not small (though the TCB exposes a minimal interface, namely a virtualized interface to Web Workers; see Section 3). Second, despite our best efforts, the guest code sometimes needs minor restructuring; however, the required code changes are few and easy to make (see Section 6). Third, while Web Workers are available in recent browsers and expected to become ubiquitous, we are currently in a transition period (see Section 2.4).

There is a lot of related work, and we cover it in detail in Section 7. For now, we just note that no other work that we are aware of virtualizes the browser in a backward compatible way, requires no server or domain configuration, and protects against resource exhaustion attacks. The contributions of this work, then, are as follows:

- Applying the operating systems ideas of sandboxing, virtualizing, and resource management to JavaScript.
- The design of TreeHouse, which instantiates these OS ideas without browser modification.
- The implementation and evaluation of TreeHouse.

2 Background

This section explains the aspects of the Web browser ecosystem that are relevant to TreeHouse.

2.1 Some details of modern Web browsers

A Web page is a ***document*** composed of HTML markup, CSS styles, and JavaScript (JS) code. HTML describes the structure and content of the document, CSS describes its visual presentation, and JavaScript [18] adds dynamic behavior. Browsers provide an API through JavaScript, called the ***Document Object Model (DOM)***, which represents the page as a tree of nodes with methods and properties. Scripts within a Web page use the DOM to examine and change the page.

The browser also exposes an API through JavaScript that provides network access, multimedia capabilities, file access, asynchronous interrupts, and local storage. A notable class in this API is ***XMLHttpRequest (XHR)***, which allows a script to make an HTTP request. The browser restricts such requests, allowing them only to the document's ***origin***, a tuple of (scheme, domain, port).[3]

This restriction is part of the ***Same Origin Policy (SOP)***, whose purpose is to contain information leaks. Consider a user with the authority to get data from a restricted site. If such a user visits a site with malicious scripts that issue XHRs as the user to the restricted site, then, in the absence of the SOP, the browser would permit the XHRs; the scripts could thus wrongly extract data and send it to the malicious site. The SOP prevents such leaks by regarding each origin as a separate security principal and then preventing the browser from becoming a channel that leaks information among principals.

Because of this model, scripts in documents from the same origin may access the DOMs of each other's documents but not the DOMs of documents from any other origin. Perhaps confusingly, the SOP includes exceptions to this rule for some types of content. For example, a document is permitted to include and execute scripts from other origins. However, the origin that the cross-origin script is assigned by the browser is the origin of the including document, *not* the origin from which the script

[2] Others have virtualized this interface, in the browser [14, 20, 32, 40, 41] and on the server [3, 17], but with goals different from ours (§7).

[3] This tuple is drawn from the document's URL; for example, the origin corresponding to https://www.example.com:1234/foo/bar.html is (https, www.example.com, 1234).

was downloaded. For example, if a page from foo.com includes a script from bar.com, the browser allows the script to access content from foo.com but not bar.com.

2.2 JavaScript

The following properties of JavaScript help TreeHouse in its goal of isolating scripts. First, a script cannot create a reference to an arbitrary memory location: a script can access only objects that it creates itself and objects that the browser hands to it. Second, JavaScript as a language provides no facilities for I/O, meaning that, with the exception of covert channels, scripts can communicate outside their environment only by using the browser's API. Finally, as of version 5.1 of JavaScript, scripts can ***freeze*** properties of objects. Once a property is frozen, further attempts to assign or delete its value have no effect.

2.3 Frames

Browsers ship with a mechanism called iframes that are intended to create a logically separate entity within an enclosing page. However, iframes do not provide the isolation that one might want. First, iframes run in the same thread as their enclosing page. If code blocks in an iframe, the whole page blocks. Second, iframes can consume resource budgets that the browser imposes on the entire page. For example, browsers limit the number of in-flight XHRs (to any origin and in total), and misbehaving code can exhaust this limit (as has been observed [8]).

2.4 Web Workers

JavaScript is single-threaded and does not support preemption. Absent further mechanism, then, a script must break up compute-bound tasks, periodically returning control to the browser's event loop, or else the page becomes unresponsive. This limitation has motivated ***Web Workers***, a recent[4] browser feature that lets documents run scripts in a "separate parallel execution environment" [2]. In all cases that we are aware of (desktops, smart phones, etc.), these separate environments are preemptively scheduled processes or threads, as provided by the underlying operating system. For computations that admit parallelism, then, Web Workers allow application developers to write code in a threaded style.

Web pages can create an arbitrary number of workers; each gets its own JavaScript environment. The origin assigned by the browser to those workers is the origin of the document that created the worker, called the ***parent document***. A worker and its parent communicate using an asynchronous message-passing facility provided by the browser (called `postMessage`). Scripts are otherwise isolated: a script in a worker cannot import a reference to an object outside the worker or export a reference to an object inside the worker. Workers also do not have access to the DOM or most other browser resources. However, they can import scripts by URL, create child workers, and issue XHRs.

The goal of Web Workers was concurrency, but TreeHouse repurposes them, as described in the next section.

[4] Web Workers are part of the HTML5 specification [2] and are supported by the latest versions of all major browsers. Internet Explorer (IE) is a special case. As of this paper's publication, IE 10.0, which supports Web Workers, is in beta and expected to ship in 2012; when it does, IE's dominance is such that Web Workers will quickly be on a large majority of desktops.

3 Design of TreeHouse

3.1 Threat model and requirements

Threat model. Our threat model assumes that an honest user interacts with a Web application using an uncompromised and correct browser. The application is written by an honest ***author***. We will assume that the following content is correct and served from uncompromised Web servers that are part of the author's trust domain: a distinguished HTML page (called the ***host page***), a distinguished set of JavaScript, and any CSS and JavaScript directly included in the host page. The adversary can control the contents of any JavaScript, HTML or CSS that is downloaded from a server not under the author's control (for example, the adversary can supply the ad or framework code); this content is ***untrusted*** by the author, meaning that neither the author nor TreeHouse can depend on the correctness of this content. For prudence, an author aware of his imperfections (Dr. Jekyll) may wish to regard code that he himself wrote as being supplied by the adversary (Mr. Hyde), trusting only a minimal host page and the distinguished JavaScript.

Requirements. The design of TreeHouse is driven by the following requirements.

- *Isolate untrusted content.* Before anything else, TreeHouse needs a mechanism for *isolating* content. (We make this notion more precise below.) Such content would ideally have no impact on the execution of TreeHouse or on the rest of the application, and limited impact on their performance.
- *Interpose on untrusted content.* It is not sufficient for TreeHouse simply to isolate content. To perform useful work, content needs to communicate with the application, to affect the browser, and to consume browser resources. That is, the untrusted content may need to interact with the document's DOM, to gain access to cookies or files, to create Web Workers, and to consume outbound network requests and local storage. However, this impact needs to be controlled. Thus, TreeHouse must interpose on attempts by untrusted content to do useful work, to decide which attempts are permissible. This will allow TreeHouse to protect integrity (for example, by forbidding unauthorized

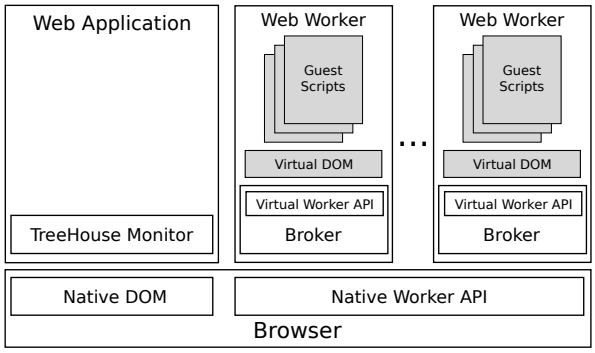


Figure 1—Architecture of TreeHouse. Shaded portions are untrusted by the application.

modifications to the DOM), availability (for example, by disallowing scripts from over-consuming required local storage), and privacy (for example, by disallowing a script from consuming an outbound network request to another origin, since it could use such a request to leak data).

- *Manage resources at fine grain.* TreeHouse must provide a way for application authors to express what access is permissible by guests, and what resources guests may consume. Consistent with the principle of least privilege [48], the granularity of permissions should be as fine as possible.

3.2 Overview of TreeHouse

Figure 1 depicts TreeHouse. For isolation, TreeHouse runs untrusted code in Web Workers (§2.4); once the code is running under TreeHouse in the Web Worker, we call it ***guest code*** or ***sandboxed code***. For interposition, TreeHouse installs a ***broker*** in each worker that virtualizes the browser's resources. For example, the broker exports to the worker a ***Virtual DOM (VDOM)*** that looks to guests like the browser's API. Interposition also requires a ***monitor*** that runs in the JavaScript environment of the window or tab in which the user loaded the application. The monitor applies guests' VDOM modifications to the real DOM, and delivers DOM events to guests—if permitted. What is permitted (regarding the DOM and access to other browser resources) is decided by the application author, and the definition of this policy is the only application customization; the monitor is the same across Web applications. Communication between the guest script and the monitor is handled by the broker, using message passing (§2.4). For example, the broker translates VDOM changes into messages to the monitor.

The rest of this section details the isolation mechanism (§3.3), interposition and virtualization (§3.4), and how application authors express policy (§3.5).

3.3 Isolation

We say that script *B* is ***isolated*** from script *A* if (1) *B* cannot prevent *A* from running; and (2) *B* cannot access *A*'s JavaScript environment. TreeHouse applies this notion to isolate untrusted scripts from the monitor and other application code. In the rest of this section, we describe how TreeHouse enforces (1) and (2). In short, TreeHouse uses Web Workers, which is perhaps surprising, since they were introduced for a different purpose.

For condition (1), each Web Worker runs in its own preemptively scheduled thread (§2.4), so the ability of a script inside a worker to affect the liveness of code outside it is restricted by the scheduling policy of the operating system. For example, if code in the worker enters an infinite loop, the performance of the system degrades but not to the point of preventing application code *outside* the worker from making progress.

For condition (2), we note that in a browser that correctly implements Web Workers (as assumed by our threat model), each worker has its own JavaScript environment. Moreover, JavaScript code cannot construct a reference to an object outside its environment (§2.2), and the only communication mechanism available to Web Workers, `postMessage` (§2.4), does not pass references.

Thus, to isolate a script, it is sufficient to run that script in a worker; this isolates the script from the monitor and from scripts in other workers.

3.4 Interposition and virtualization

We now motivate and describe TreeHouse's interposition mechanism. Interposition is needed for two reasons.

The first reason is that isolating code in the sense above does not prevent it from doing harm. Consider a Web application that allows users to upload and retrieve photos, and to send and receive private messages. Assume for illustration that these two functions are implemented by two logical services at the application's origin. If a script in the application is "supposed" to invoke only the photos API, then, by the principle of least privilege, that script should in fact be limited to the photos API. However, a malicious script—even if isolated—can still invoke the messages API and thus forge messages from the user. The script can also leak messages, as follows. Any origin can import scripts from any other (as discussed in Section 2.1 in the context of the SOP). If the malicious script has gained access to the user's private messages by invoking the messages API, the script can then exfiltrate a message by "importing" a script from http://evil.com/script.js?private-message, thereby leaking the private-message to evil.com.

To prevent attacks like the ones above and others, TreeHouse requires some logic between the isolated code and the outside world. This interposing logic must protect not only privacy but also integrity (by disallowing

unauthorized changes to the page seen by the user) and availability (by limiting the consumption of resources).

This brings us to the second reason that interposition is needed: some of the third-party actions need access to actual browser resources (the DOM, XHRs, etc.) to get work done. Thus, the interposing logic must not only intercept various actions but also decide which of them are permissible.

Questions of interface and mechanism. There are now three interrelated questions that TreeHouse must answer:

(1) Through what interface should guest code request resources from TreeHouse's interposing logic?

The challenge here is that scripts' resource requests need to be clear to TreeHouse (so it can decide whether to grant the request), yet the goal of deployability (§1) means that the script should not be altered to make requests through a new interface. TreeHouse's response to this challenge is reminiscent of trap-and-emulate [4] in the virtual machines context: TreeHouse *virtualizes* the browser's API and arranges to be invoked whenever a script requests access to a browser resource, whether that resource is one in the main application or in a Web Worker. But we now have a second question:

(2) What mechanism(s) should TreeHouse use to interpose and virtualize?

The challenge here is that JavaScript has a wide interface to the browser. This interface is narrower in Web Workers but still not so very narrow. Web Workers can access information about the application (such as its URL), import scripts, make network requests, and create child Web Workers. In some browsers, workers can use local storage as well as any files that the user has permitted the application to access. Meanwhile, TreeHouse needs to interpose on all of these actions. This brings us to the third question:

(3) How should application authors express policy so that TreeHouse can decide whether to grant or deny a given resource request?

The challenge here is ensuring a natural map between the language in which application authors express permissions and the implicit requests made by scripts. TreeHouse's approach here is to require application authors to express permissions in terms of the browser's API: the application author expresses which methods (and the arguments to those methods) guest code can use. Section 3.5 provides the details of policy expression. The rest of this section takes such a policy as a given and details TreeHouse's response to question (2).

Virtualizing the browser's API. Recall that TreeHouse's approach is reminiscent of trap-and-emulate. We first describe how TreeHouse arranges for traps and then how it performs the "emulate".

When TreeHouse creates a worker, it loads into the worker a script, called a ***broker***, which is part of TreeHouse's trusted computing base (see Figure 1). The broker must (a) interpose on calls to the browser's API that are available in workers (issuing XHRs, etc.) and (b) create a virtual DOM and interpose on interactions with it (the DOM itself is not available in workers; see Section 2.4). For (a), before the guest code loads, the broker modifies the worker's environment to wrap each function in the browser's API with a new implementation that interposes the broker when the function is called. Specifically, the broker uses JavaScript's `Object.defineProperty` API to associate the function name with a new function, and to freeze (§2.2) the association between the name and the new function, which prevents guest code from undoing this environment manipulation. For (b), the broker constructs a VDOM (§3.2), which contains subtrees of the real DOM (the application decides which subtrees). The VDOM implementation that TreeHouse uses (§5) raises events when the guest modifies the VDOM, and the broker registers a handler for such events.

For the "emulate" piece, there are two flows to consider: guest invocations of the browser API and event delivery from the main browser to the guest. When guest code invokes the browser's API or modifies the VDOM, the broker, being interposed, first applies the application's access control policy, to decide whether the guest action is permitted. If it is not permitted, the broker terminates the guest. If it is permitted, then, with the exception of DOM changes and asynchronous API methods for which a native implementation is not available in a Web Worker, the broker completes the call itself. We note that completing the call may involve further interposition—on the return value. For example, if the return value is an object with methods, then the broker replaces those methods with functions that interpose the broker.

In the case of DOM changes and asynchronous API methods, the broker delegates the request to the original browser API; to do so, the broker serializes the request and passes it to the monitor using `postMessage`. The monitor then makes the DOM modification or completes the API call. (We describe below how TreeHouse handles the mismatch between the synchronous interface to the VDOM and asynchronous `postMessages`.)

Event delivery is similar. If guest code wishes to register to be notified of a DOM event, then the broker receives the registration request and notifies the monitor. The monitor then registers its own *generic* handler in the application's DOM; when the event fires, the monitor notifies the broker, which re-raises the event in the VDOM on the handler registered by the guest.

As a final detail concerning virtualization, we note that the broker, when interposed, must sometimes do more

than check permissions. As an example, we briefly consider the API to create Web Workers. If the guest (itself in a Web Worker) attempts (and is permitted) to construct a Web Worker, the broker invokes the browser's API to construct a new Web Worker, and returns an object that wraps that new worker. To maintain interposition in the new worker, the broker, which is now a ***parent broker***, runs a ***child broker*** in the new worker. The child broker is identical in function to the parent except that, by default, it virtualizes only the interface that browsers expose in a Web Worker (no VDOM, etc.), as this is what a script that is *intended* to be run in a Web Worker would expect. The parent broker relays messages between the child broker and the monitor. TreeHouse virtualizes outbound network requests, file access, and local storage similarly.

A sync-vs-async mismatch, and some limitations. TreeHouse's approach to virtualizing the browser's API sometimes requires that the broker present a blocking interface; meanwhile, completing such a function or method call may require that the broker send an asynchronous `postMessage` to the monitor—and that the broker then return to the event loop so that it can receive the reply event (§2.4). The mismatch here is between a synchronous interface and an event-driven implementation, and there are two broad cases.

First, if the guest call can be "faked" by the broker, then the broker can present a synchronous interface and carry out the request asynchronously. For example, the broker presents a blocking interface to the VDOM and propagates changes to the application's DOM asynchronously. However, now TreeHouse must handle the equivalent of concurrent threads (the workers) sharing memory (the main DOM), where the threads have caches of that memory (the VDOMs in each thread). For simplicity, TreeHouse's response is to prevent sharing altogether: the monitor guarantees that a DOM node exists in at most one VDOM at once.[5] We do not believe that this limitation will be onerous for application authors.

The second case is when the guest call cannot be faked by the broker. As an example, consider `window.alert`, which creates a dialog box that blocks the calling script. No alert method is available in Web Workers, so for the broker to display an alert, it would have to send a request to the monitor and then return to the browser's event loop to await the reply—which conflicts with the guest's expectation of a blocking call. TreeHouse does not handle this case; if guest code calls such a method, it fails with a runtime error. Fortunately, there are few such methods, and they are rarely used by third-party code.

[5] An alternative approach would be to allow resources such as DOM nodes or cookies to appear in workers with either exclusive read-write access, or shared read-only access.

```
1   '!api': {
2    'XMLHttpRequest': {
3     '!invoke': true,
4     '!result': {
5      // permit only asynchronous XHRs
6      open: function (verb, url, async) {
7       return async === true;
8      },
9
10     '*': true // default rule
11    }
12   }
13  }
```

Figure 2—The portion of TreeHouse's base access control policy that governs XHRs. The policy forbids synchronous XHRs.

3.5 Resource control policy

The application author manages the guest's access to resources by (a) deciding which DOM elements to place in the guest's VDOM and (b) expressing policies that govern the guest's interaction with the browser's API (including the VDOM). This section details the second aspect; the next section gives examples of both aspects.

At a high level, the application author expresses access control policy in terms of the browser's API: what calls are permitted, and what arguments to those calls are permitted. In more detail, the author creates a per-guest JavaScript ***policy object*** and hands this object to TreeHouse (see Section 5 for the details of this hand-off). To simplify slightly, the policy object implements a key-value map from browser API elements to permissions: the keys name browser API elements,[6] and the values are ***rules***. A rule can be a Boolean value, a function, or a regular expression. If the rule is the Boolean value True, then the guest is permitted to invoke the given method or set the given property. If the rule is a function, the broker, at permission check time, executes the function (which should evaluate to a Boolean) to decide whether the action is permitted. If the rule is a regular expression, then it refers to a property; in this case, the guest is permitted to set the given property to a value v if v matches the regular expression.

TreeHouse has a ***base policy*** that authors are not supposed to override. This policy is there for their protection (and TreeHouse's). For example, as depicted in Figure 2, the base policy specifies that guests must open XHRs (§2.1) asynchronously; otherwise, a guest could prevent the broker from running. For a given action by a guest to take place, it must be permitted by *both* the per-guest policy and the base policy.

[6] The "keys" are structured hierarchically (reflecting the browser API's hierarchy), and can include wildcard components. This way, the author can use one rule to restrict the guest's use of an entire subtree of the hierarchy (say multiple API calls or elements).

```
1  var xhr = new XMLHttpRequest();
2
3  // initialize a request to get a list of the
4  // first ten photos
5  xhr.open('GET',
6   '/api/photos?start=0&count=10', true);
7
8  // register a callback to be notified when
9  // the XHR completes
10 xhr.onreadystatechange = function (e) {
11  if (xhr.readyState === 4) {
12   if(xhr.status === 200) {
13    console.log(xhr.responseText);
14   } else {
15    console.log("Error fetching photos");
16   }
17  }
18 };
19
20 xhr.send(null); // start the request
```

Figure 3—Example code to issue an XHR (§2.1) to a Web service that delivers photos.

```
1  '!api': {
2   'XMLHttpRequest': {
3    '!invoke': true,
4    '!result': {
5
6     // permit GET requests to resources on
7     // the application's origin whose URLs
8     // begin with '/api/photos'
9     open: function (verb, url, async) {
10     return verb === 'GET' &&
11       url.indexOf('/api/photos') === 0;
12    },
13
14    '*': true // default rule
15   }
16  }
17 }
```

Figure 4—Policy that gives limited permission: guest code can issue XHRs freely but only to particular services.

TreeHouse ships with a ***default*** or ***reference policy***, which whitelists unprivileged operations but denies everything else. For example, the reference policy forbids opening XHRs. This policy is 355 lines of code, including comments (and excluding unnecessary blacklisting for documentation). Overriding the reference policy need not be complex; we expect a typical policy to require 10–100 lines of code and no more than a few hours of work from the application author (see Section 6).

4 Examples

In this section we give several example uses of TreeHouse. Our first example illustrates how access control policies interact. We then describe containing advertisements and containing third-party widgets. Finally, we show how TreeHouse can protect a hypothetical plugin architecture (for example, by preventing exfiltration).

Limiting network access. Consider an author who—wishing to employ the principle of least privilege in the design of her application—breaks it into mutually distrusting components, each running in a TreeHouse sandbox. One such component is a script that displays a slideshow widget. The script obtains a list of photos to display from a Web service exposed on the application's origin and then renders the slideshow.

Figure 3 shows the guest's code for obtaining the list of photos. Because the default policy forbids constructing an XHR object, the code would fail at line 1. Thus, the author must override the default policy to give the guest code limited permission; Figure 4 depicts an example. Now when the broker intercepts the XHR constructor, it sees that `!api.XMLHttpRequest.!invoke` is True in both the guest policy and the base policy. The broker thus permits the construction and returns a wrapped XHR, as described in Section 3.4.

When the script calls the wrapped XHR's `open` method (lines 5 and 6 of Figure 3), that call succeeds, since the relevant policies permit the arguments to `open`. In more detail, when the script calls `xhr.open`, the broker checks the `!api.XMLHttpRequest.!result.open` property in the guest's policy object. The value of that property is a function (lines 9–12 of Figure 4), and TreeHouse calls it with the arguments provided by the guest script. The function returns True (because the URL is policy-appropriate), so TreeHouse then checks the base policy (lines 6–8 of Figure 2), which also returns True (because the guest has specified an `async` XHR).

The guest can also set `onreadystatechange` and call `send` (lines 10 and 20 of Figure 3) because the guest and base policies allow this through their default rule of `!api.XMLHttpRequest.!result.*`. Of course, the author could exert more fine-grained control by creating rules for properties and methods individually.

Containing advertisements. The provenance of advertisements (ads) is often murky: sites generally display ads by delegating a piece of their page to an ad network, which may populate the space or redirect to another ad network, and so on. Indeed, ads routinely misbehave [8, 57]. For these reasons, the best practice for a site selling space is to isolate an ad. Unfortunately, today's mechanism for such isolation, the iframe, does not eliminate attacks on availability (see Section 2.3). Under TreeHouse, in contrast, the application author can schedule and limit XHRs. For example, the author can associate `!api.XMLHttpRequest.!result.open` with

```
<script src="tetris.js"
 type="text/x-treehouse-javascript"
 data-treehouse-sandbox-name="worker1"
 data-treehouse-sandbox-children="#tetris"
></script>
<script src="tetris-policy.js"
 type="text/x-treehouse-javascript"
 data-treehouse-sandbox-name="worker1"
 data-treehouse-sandbox-policy
></script>
```

Figure 5—Example `script` tag. The depicted block specifies that `tetris.js` should run in a TreeHouse sandbox and that `tetris-policy.js` is the sandbox's policy.

a function that permits the method call only if the number of outstanding XHRs from the worker is below a given threshold. Or, given suitable hooks into the wrapping machinery, the application author can queue the `open` calls, sending them one at a time.

Containing widgets. Consider a set of widgets, each of which displays a one- to five-star rating next to an entry in a product list (e.g., [7]); such widgets are typically driven by a single script that runs in the application's page. For prudence, the application developer would like to limit the widgets' influence. (The developer cannot rely on iframes because, for reasons of layout, each widget would be in a separate iframe. With n products and thus n widgets, performance would suffer.) Under TreeHouse, the developer sandboxes the widget script and sets its VDOM to include only the locations in the document where the script should display ratings widgets. This pruned DOM can be constructed programmatically, using JavaScript functions that manipulate the DOM.

Avoiding exfiltration. Consider a webmail service, ExampleMail, whose authors want to allow third-party developers to create plugins. In the status quo, such plugins would be an unacceptable security risk, as the plugins would be able to read mail and then exfiltrate it. Under TreeHouse, ExampleMail's authors can sandbox a plugin and grant it limited access to the text of emails, while preventing it from exfiltrating email. Take, for instance, a plugin that displays the word count of the currently selected message in the user's inbox. This plugin receives a VDOM that includes the email that the user is viewing together with a *display element*, where the plugin author will display the word count. The application's policy prevents network access (by disallowing access to XHRs, WebSockets, and those attributes of DOM nodes that can have a URL as their value, such as `src`) and rejects DOM changes unless the change is to a node that descends from the display element. At that point, the plugin has access to the current message and can display the word count, but it cannot exfiltrate or alter the message.

method	**description**
start()	start the sandbox
terminate()	terminate the sandbox
addChild(node)	add a node to the VDOM
removeChild(node)	remove a child from the VDOM
addEventListener(type, function)	handle events from the sandbox
postMessage(message)	send a message to the sandbox
jsonrpcCall(method, args...)	make RPC call
jsonrpcNotify(method, args...)	make RPC call (no return value)
onPolicyViolation(function)	handle policy violations
setPolicy(policy)	set the access control policy

Figure 6—API for managing TreeHouse sandboxes.

component	lines of JavaScript
Monitor	350
Broker	349
Shared by monitor and broker	369
Access control policy	794
jsdom [17]	127,652

Figure 7—Lines of code in TreeHouse.

5 Integration and implementation

To integrate TreeHouse into a Web application, the author includes the TreeHouse monitor as traditional JavaScript inside the application page. There are two ways to sandbox a script. First, the author can include a `script` tag of type `text/x-treehouse-javascript` in the application's HTML, as illustrated in Figure 5. In this case, the browser creates a node for the script in the DOM but does not execute it (because it does not recognize the script type). When the browser loads the page, the monitor finds all such tags and creates sandboxes as appropriate. In this case, the author specifies the configuration options as attributes of the `script` tag; these options include which DOM subtrees the script may access, which worker to load the script into, and which policy applies to the script. The author's other choice is to invoke an API provided by TreeHouse, allowing the author to explicitly create a sandbox, set policy, etc. This API is depicted in Figure 6.

We have run TreeHouse successfully on Chrome, Safari, Firefox, and IE10. It is implemented in 1862 lines of JavaScript plus 127,652 for jsdom [17]. Jsdom is a *server*-side DOM implementation that we modified (with several hundred lines of code) to implement the VDOM. Figure 7 gives the breakdown (according to [42]). Besides jsdom, we use the underscore (v1.1.7) and RequireJS (v0.26.0) libraries for JavaScript utilities. We have not completed cookie virtualization or VDOM implementations of the new elements and API methods that the recent HTML5 standard introduces; this is future work.

	TreeHouse slowdown (×)			
benchmark	Chrome	Firefox	IE	Safari
dom-attr	32	39	9	—
dom-modify	15	72	21	120
dom-query	2600	8000	780	—
dom-traverse	7	14	1	12

Figure 8—TreeHouse overhead on Dromaeo DOM benchmarks reported as the geometric mean, over all benchmarks in a category, of TreeHouse's speed as a multiple of the baseline's. Empty entries result from browser incompatibilities or bugs. TreeHouse adds considerable relative overhead for DOM operations, but the absolute numbers are not high; see text.

	μ (σ) page load latency (ms)			
Experiment	Chrome	Firefox	IE	Safari
DOMTRIS, baseline	24 (8)	12 (1)	6 (3)	5 (1)
DOMTRIS, TreeHouse	361 (46)	181 (4)	405 (18)	166 (34)
118KB page, baseline	25 (3)	5 (1)	11 (5)	22 (5)
118KB page, TreeHouse	976 (38)	880 (18)	1229 (66)	779 (12)
Sandbox but no VDOM	350 (53)	136 (3)	323 (13)	132 (4)

Figure 9—Page load latency, with and without TreeHouse. TreeHouse's setup costs include a fixed cost from sandboxing and a cost for VDOM population that varies with VDOM size.

6 Evaluation

To evaluate TreeHouse, we answer two questions: (1) What is the latency overhead from TreeHouse? and (2) How easy is it to incorporate TreeHouse into an application? We answer both questions by experimenting with various benchmarks and Web applications.

Our experiments run on a MacBook Pro with a 2.66 GHz Intel Core 2 Duo processor and 4 GB of RAM, running Chrome 18.0.1025.168, Firefox 10.0.2, IE 10.0.8250.0, and Safari 5.1.5 (7534.55.3). We run Internet Explorer in a Windows 8 Consumer Preview (build 8250) guest on VirtualBox 4.1.12.

Benchmarks. The Dromaeo benchmark suite [1] is large (188 benchmarks) and diverse. Its benchmarks either do not access the DOM (*non-DOM benchmarks*) or else hammer it (*DOM benchmarks*). We expect that the non-DOM benchmarks would not experience significant slowdown under TreeHouse whereas the DOM benchmarks would run slower (because TreeHouse intercepts only DOM modification). To experiment, we run the suite with and without TreeHouse (which requires minor changes to Dromaeo, to report results via `postMessage`), performing 10 runs for each of the aforementioned browsers. Our expectations about the non-DOM benchmarks mostly hold: some benchmarks run slower (2–7×, depending on the benchmark and the browser) with TreeHouse, and some run faster, though we are still investigating to understand the slowdown and the speedup. For the DOM benchmarks, our expectations also hold; we now delve into those results.

To organize the DOM benchmarks, we divide them into four categories: *dom-attr*, which stresses setting attributes on DOM elements; *dom-modify*, which stresses inserting and removing DOM elements; *dom-query*, which stresses DOM searches; and *dom-traverse*, which stresses DOM tree traversals. Figure 8 reports the results, in terms of the geometric mean of each category's overhead. As expected, TreeHouse imposes significant overhead on DOM operations. The largest overhead (by far) is in DOM queries, and this overhead is staggering.

However, the operations that TreeHouse has blown up are not expensive in absolute terms or likely to be executed often. For example, depending on the browser, TreeHouse imposes overhead of 13,000–120,000 on `getElementsByTagName` (which returns all DOM nodes of a particular type, such as IMG) when searching for a non-existent type. Yet even after the blowup, each call to `getElementsByTagName` takes slightly less than 1 ms. Moreover, a best practice in Web application development is to avoid DOM queries (by caching). We expect applications that follow this practice not to be significantly slowed under TreeHouse. Of course, applications that do not follow this practice would need changes to run efficiently under TreeHouse.

Latency of page load. TreeHouse has two setup costs: *sandbox setup* (loading the guest into a worker, interposing the broker, etc.) and *VDOM population*. To assess both costs, we measure page load latency for DOMTRIS [50] (a JavaScript Tetris clone that uses the DOM to render the game and handle user input; the choice of application is borrowed from [40]) and a large Web page, with and without TreeHouse.

To measure the page load latency, we include two scripts, one in the page's header and one at the end of its body, measuring the time elapsed between each. When we measure under TreeHouse, the second script waits for a message from the guest, indicating that VDOM population is complete. This approach exploits the fact that the browser guarantees to execute the second script only after the entire DOM is parsed. We perform 10 runs in each browser, collecting timing data from JavaScript's `Date.now()`, which reports time in milliseconds.

Figure 9 reports the results with and without TreeHouse. For DOMTRIS, the average overhead of TreeHouse (TreeHouse row minus baseline row) is 161–399 ms, depending on the browser (here and below, the range reflects the minimum and maximum over the four browsers in our experiments). For the large Web page, the average overhead of TreeHouse is 757–1218 ms. We hypothesize that the larger overhead in this case derives from the size of the Web page's VDOM, which translates to higher VDOM population costs. To try to separate the sandboxing and VDOM costs, we run an experiment that

starts a TreeHouse sandbox with no VDOM; the average cost is 132–350 ms. TreeHouse's remaining costs come from application overhead and populating the VDOM.[7]

Usability for developers. TreeHouse requires two kinds of effort from developers: porting to TreeHouse and writing policies. We briefly assess each.

To port DOMTRIS to TreeHouse, we had to change only 28 lines of code in DOMTRIS. We are also in the process of porting several frameworks to TreeHouse; this effort both enhances TreeHouse's usability (by letting developers run existing framework-based applications in TreeHouse) and gives a sense of the work required to port a complex application to TreeHouse. So far, with 2 extra lines of code in the Zepto framework [67], all but three of the Dromaeo benchmarks that target Zepto's interface (consisting of 28 microbenchmarks) run successfully against Zepto-in-TreeHouse in Chrome and Safari. With changes to 18 lines of code in the Prototype framework [45], all but three of the Dromaeo benchmarks that target the Prototype interface (consisting of 29 microbenchmarks) run successfully against Prototype-in-TreeHouse in Chrome, IE, and Safari.

To evaluate the effort required to write real policies, we designed a sample policy for the ExampleMail plugin described in Section 4. The policy contains 41 lines of code and took approximately 30 minutes to write.

Summary. TreeHouse imposes significant overhead for "privileged operations", but, unless the application spends most of its time in such operations, total overhead should be far lower. TreeHouse also adds to initial page load latency, particularly when the VDOM is large. Porting applications to TreeHouse appears to require only modest effort, as does writing a non-trivial policy.

7 Related work

We survey related work by covering current browsers, isolation by frames, browser modification and redesign, language-based approaches, and related mechanisms.

Current browsers. The OP browser [24] creates a new process for each document. The Chrome browser, based on the Chromium project [5, 47], and Internet Explorer [65, 66] implement similar forms of isolation. These mechanisms, sometimes referred to as *process-per-tab*, allow the browser to contain site crashes and continue running. These mechanisms are orthogonal to TreeHouse: they isolate Web applications from each other but do not isolate scripts *within* a Web application.

[7] Our experiments do not let us determine the VDOM population costs precisely. We want to solve for V in $T = B + S + V$, where T is the TreeHouse results, B is the baseline results, and S is the sandboxing cost. Unfortunately, we observe T and B but not S. Instead, our experiment observes $S + E$, where E includes costs that B also includes. Thus one can derive a range for V, by varying E between 0 and B.

Frame isolation. Some schemes isolate JavaScript by using two browser features: the Same Origin Policy (§2.1) and iframes (§2.3). For example, SMash [31], Subspace [28], and OMOS [64] isolate scripts and components by running them in iframes served from different origins and then provide a mechanism for the iframes to communicate. AdJail [32] runs untrusted advertisement code in an iframe and then replicates changes to that iframe's DOM back to the main page.

All of these schemes contrast with TreeHouse as follows. First, unlike TreeHouse, they require the application to have a unique origin per isolated component. Second, as mentioned in Section 2.3, scripts in an iframe can interfere with the application's liveness, by going into an infinite loop or consuming resources; TreeHouse, in contrast, tolerates these cases (see Section 3.3). Third, these systems do not prevent the kind of information leaks detailed in Section 3.4 (their iframes can continue to "import" content from arbitrary locations).

Browser modification and redesign. We first summarize work that proposes browser re-architecture or modification and then explain why we avoided such changes.

The Atlantis browser [39] lets Web sites define their own layout, rendering, and scripting engines. Atlantis defines a small set of primitives and exposes them to sites. Applications can use Atlantis's primitives to achieve TreeHouse's goal: sandboxing scripts.

In contrast, other proposals for new browsers protect Web sites from each other but do not isolate scripts within a site. As examples, the Tahoma browser [9] isolates Web applications in virtual machines, the Illinois Browser Operating System (IBOS) [53] proposes an operating system and browser that map browser abstractions to hardware abstractions, and the Gazelle browser [60] treats origins as principals in a multi-principal operating system, isolating their content in restricted or sandboxed OS processes.

Some browser extensions have goals that overlap with those of TreeHouse. MashupOS [59] makes the browser a multi-principal operating system for Web applications. BEEP [29] lets Web sites restrict the scripts that run in each of their pages. ConScript [38] enforces application-specified security policies. OMash [10] restricts communication to public interfaces declared by each page. BFlow [62] adds information flow tracking to the browser, allowing untrusted JavaScript to operate on private data without compromising confidentiality.

The above projects partially inspire TreeHouse. However, they require browser changes. Meanwhile, if a browser change requires application changes (and all of the above proposals do), authors must either wait for all supported browsers to make the change or maintain two versions of their application. And an author who supports

an enhancement available in one or two browsers before it is adopted elsewhere takes the risk that other browser vendors choose not to add the feature at all. This risk is not small: browser vendors have historically been reluctant to implement new security features [27].

Language-based approaches. One way to isolate JavaScript is to constrain it to a subset of the language. Before this code is sent to the browser, it passes through a server-side verifier. Various projects apply this high-level approach, including Caja [41], FBJS [20], ADSafe [11], Browsershield [46], work by Maffeis et al. [34, 35], and work by Barth et al. [6]. JSReg [25] uses regular expressions to rewrite untrusted scripts.

The primary disadvantage of a restricted subset is that few libraries are written in them. Moreover, these subsets preclude many popular JavaScript idioms (e.g., the `this` keyword), making programming more difficult. While the designers of these approaches have gone to great lengths to ease the porting burden, the process of translating from arbitrary JavaScript to the restricted subset still requires manual work.

Language-based approaches have several other disadvantages. First, the application must serve the verified code from a server under its control. As a result, applications cannot gain from the performance advantages of content distribution networks (CDNs), as mentioned in the Introduction. Second, many language isolation techniques employ runtime logic that interposes on *all* object property accesses, which imposes non-trivial overhead. TreeHouse, by contrast, can isolate scripts from any origin and interposes only on privileged operations.

Related mechanisms. Several projects use Web Workers, virtualize the DOM, or create containers for isolated code. Some of these projects have inspired TreeHouse, but none shares all of our goals.

JSandbox [23] and Bawks [55] prototype a low-level IPC mechanism by which application code can load code into a Web Worker and allow callbacks. They do not provide virtualization (§3.4) or resource control (§3.5).

TreeHouse borrows DOM virtualization from previous work with different goals. Jsdom [17] (whose implementation TreeHouse uses) and dom.js [3] aim to provide a convenient toolkit for manipulating Web pages; they are geared to environments such as server-side JavaScript, where there is no "native" DOM. Mugshot [40] virtualizes part of the client's DOM but does so to create a replay system; it does not isolate scripts. AdSentry [14] is geared to isolation and, like TreeHouse, it interposes on DOM operations, applies access control policies, and delegates to the native DOM. Unlike TreeHouse, however, AdSentry requires browser modification and does not protect the application's liveness: if the ad blocks, so does the application.

In concurrent work, js.js [54] takes a different approach to sandboxing. The authors implement a JavaScript interpreter in JavaScript; the interpreter runs untrusted code and exposes no privileged methods or properties by default. While js.js and TreeHouse share some goals, the performance characteristics are different: DOM changes in js.js run at "native speed", but everything else runs two orders of magnitude slower than native. Under TreeHouse, DOM changes are expensive, but everything else runs roughly at native speed.

Finally, TreeHouse is inspired by classic approaches to isolation. Traditional sandboxing [21, 22, 36, 49, 51, 58], applied to browsers by Native Client [61] and Xax [15], contains legacy x86 code that expects to interact with the system call interface or machine resources. TreeHouse, however, contains legacy JavaScript code that expects to interact with the browser's resources.

8 Discussion and conclusion

We briefly consider how future help from browsers could address TreeHouse's limitations. First, the blocking-vs-event-driven mismatch and its consequences (§3.4) could be addressed if browsers exposed a way for JavaScript code to receive a message *synchronously*. Second, TreeHouse relies on the assumption that a worker cannot access the application's DOM; thus, it would be a boon to TreeHouse if browsers standardized the interface that is visible within workers. Of course, the standardization that would most address TreeHouse's performance and compatibility limitations would be incorporating TreeHouse—or functionality like it—into browsers.

Even if browser standardization does not come to pass, we believe that TreeHouse is promising: it is a practical, deployable, and usable way to give Web application authors fine-grained control over untrusted JavaScript code.

Acknowledgments

Insightful comments by John Hammond, Dave Herman, Jon Howell, Donna Ingram, James Mickens, Emmett Witchel, the anonymous reviewers, and our shepherd, Sam King, substantially improved this draft. This research was partially supported by AFOSR grant FA9550-10-1-0073 and by NSF grants 1055057 and 1040083.

TreeHouse is housed at `https://github.com/lawnsea/TreeHouse`. The site includes the code that implements TreeHouse, the pages used in our experiments, and demos.

References

[1] Dromaeo: JavaScript performance testing. `http://dromaeo.com/`.

[2] HTML5 living standard. `http://www.whatwg.org/specs/web-apps/current-work/multipage/`.

[3] A. Gal et al. dom.js. `https://github.com/andreasgal/dom.js`.

[4] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS*, 2006.

[5] A. Barth, C. Jackson, C. Reis, and the Google Chrome Team. The security architecture of the Chromium browser. http://seclab.stanford.edu/websec/chromium/chromium-security-architecture.pdf, 2008.

[6] A. Barth, J. Weinberger, and D. Song. Cross-origin JavaScript capability leaks: Detection, exploitation, and defense. In *USENIX Security*, 2009.

[7] http://www.bazaarvoice.com/.

[8] J. Bixby. Fourth-party calls: What you don't know can hurt your site. . . and your visitors, July 2011. http://www.webperformancetoday.com/2011/07/14/fourth-party-calls-third-party-content/.

[9] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen. A safety-oriented platform for web applications. In *IEEE Symp. on Security & Privacy*, 2006.

[10] S. Crites, F. Hsu, and H. Chen. OMash: Enabling secure web mashups via object abstractions. In *ACM CCS*, 2008.

[11] D. Crockford. ADsafe: Making JavaScript safe for advertising. http://www.adsafe.org.

[12] Department of Defense. Trusted computer system evaluation criteria (orange book), 1985. DoD 5200.28-STD.

[13] Dojo Team. Dojo toolkit. http://dojotoolkit.org/.

[14] X. Dong, M. Tran, Z. Liang, and X. Jiang. AdSentry: comprehensive and flexible confinement of JavaScript-based advertisements. In *Annual Computer Security Applications Conference (ACSAC)*, 2011.

[15] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *OSDI*, 2008.

[16] J. R. Douceur, J. Howell, B. Parno, M. Walfish, and X. Xiong. The Web interface should be radically refactored. In *ACM Workshop on Hot Topics in Networks (HotNets)*, 2011.

[17] E. Insua et al. jsdom. https://github.com/tmpvar/jsdom.

[18] ECMA. *ECMA-262: ECMAScript Language Specification*, 5.1 edition, June 2011.

[19] Ext JS Team. Ext JS. http://www.sencha.com/products/extjs.

[20] Facebook Team. FBJS. http://developers.facebook.com/docs/fbjs/.

[21] B. Ford and R. Cox. Vx32: Lightweight user-level sandboxing on the x86. In *USENIX Annual Technical Conference*, 2008.

[22] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *NDSS*, 2003.

[23] E. Grey. JSandbox. https://github.com/eligrey/jsandbox.

[24] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *IEEE Symp. on Security & Privacy*, 2008.

[25] G. Heyes. JSReg: JavaScript regular expression based sandbox. http://code.google.com/p/jsreg/.

[26] W. Huang. "HDD Plus" malware spread through major ad networks, using malvertising and drive-by download, Dec. 2010. http://blog.armorize.com/2010/12/hdd-plus-malware-spread-through.html.

[27] C. Jackson. Crossing the chasm: Pitching security research to mainstream browser vendors. http://www.usenix.org/events/sec11/stream/jackson/index.html.

[28] C. Jackson and H. J. Wang. Subspace: Secure cross-domain communication for web mashups. In *WWW*, 2007.

[29] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW*, 2007.

[30] jQuery Team. jQuery. http://jquery.com/.

[31] F. D. Keukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama. SMash: Secure component model for cross-domain mashups on unmodied browsers. In *WWW*, 2008.

[32] M. T. Louw, K. T. Ganesh, and V. N. Venkatakrishnan. AdJail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *USENIX Security*, 2010.

[33] T. Luo and W. Du. Contego: Capability-based access control for web browsers. In *International Conference on Trust and Trustworthy Computing*, 2011.

[34] S. Maffeis, J. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *IEEE Symp. on Security & Privacy*, 2010.

[35] S. Maffeis, J. C. Mitchell, and A. Taly. Isolating JavaScript with filters, rewriting, and wrappers. In *European Conference on Research in Computer Security*, 2009.

[36] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *USENIX Security*, 2006.

[37] L. A. Meyerovich, A. P. Felt, and M. S. Miller. Object views: Fine-grained sharing in browsers. In *WWW*, 2010.

[38] L. A. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *IEEE Symp. on Security & Privacy*, 2010.

[39] J. Mickens and M. Dhawan. Atlantis: Robust, extensible execution environments for web applications. In *SOSP*, 2011.

[40] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic capture and replay for JavaScript applications. In *NSDI*, 2010.

[41] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript, Jan. 2008. http://google-caja.googlecode.com/files/caja-spec-2008-01-15.pdf.

[42] CLOC: Count Lines of Code. http://cloc.sourceforge.net/.

[43] K. Patil, X. Dong, X. Li, Z. Liang, and X. Jiang. Towards fine-grained access control in JavaScript contexts. In *Intl. Conference on Distributed Computing Systems (ICDCS)*, 2011.

[44] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. ADsafety: Type-based verification of JavaScript sandboxing. In *USENIX Security*, 2011.

[45] Prototype Team. Prototype. http://www.prototypejs.org/.

[46] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *OSDI*, 2006.

[47] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *EuroSys*, 2009.

[48] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proc. IEEE*, 63(9):1278–1308, Sept. 1975.

[49] M. Seaborn. Plash: tools for practical least privilege. http://plash.beasts.org/index.html.

[50] J. Seidelin. DOMTRIS: A DHTML Tetris clone. http://www.nihilogic.dk/labs/tetris/.

[51] C. Small and M. Seltzer. MiSFIT: Constructing safe extensible systems. *IEEE Concurrency*, 6(3):34–41, 1998.

[52] S. Souders. Performance of 3rd party content, Feb. 2010. http://www.stevesouders.com/blog/2010/02/17/performance-of-3rd-party-content/.

[53] S. Tang, H. Mai, and S. T. King. Trust and protection in the Illinois browser operating system. In *OSDI*, 2010.

[54] J. Terrace, S. R. Beard, and N. P. K. Katta. JavaScript in JavaScript (js.js): Sandboxing third-party scripts. In *USENIX WebApps*, 2012.

[55] P. Theriault. Bawks JavaScript sandbox. http://bawks.creativemisuse.com/.

[56] https://twitter.com/about/resources/widgets.

[57] A. Vance. Times web ads show security breach. *The New York Times*, page B5, Sept. 2009. http://www.nytimes.com/2009/09/15/technology/internet/15adco.html.

[58] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP*, 1993.

[59] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for Web browsers in MashupOS. In *SOSP*, 2007.

[60] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle Web browser. In *USENIX Security*, 2009.

[61] 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 *IEEE Symp. on Security & Privacy*, 2009.

[62] A. Yip, N. Narula, M. Krohn, and R. Morris. Privacy-preserving browser-side scripting with BFlow. In *EuroSys*, 2009.

[63] YUI Team. YUI. http://yuilibrary.com/.

[64] S. Zarandioon, D. D. Yao, and V. Ganapathy. OMOS: A framework for secure communication in mashup applications. In *Annual Computer Security Applications Conference (ACSAC)*, 2008.

[65] A. Zeigler. IE8 and loosely-coupled IE (LCIE), 2008. http://blogs.msdn.com/b/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx.

[66] A. Zeigler. Tab isolation, 2010. http://blogs.msdn.com/b/ie/archive/2010/03/04/tab-isolation.aspx.

[67] Zepto.js Team. Zepto.js. http://zeptojs.com/.

Cloud Terminal: Secure Access to Sensitive Applications from Untrusted Systems

Lorenzo Martignoni,* Pongsin Poosankam,*† Matei Zaharia,* Jun Han,† Stephen McCamant,*
Dawn Song,* Vern Paxson,* Adrian Perrig,† Scott Shenker,* and Ion Stoica*
*University of California, Berkeley and †Carnegie Mellon University

Abstract

Current PC- and web-based applications provide insufficient security for the information they access, because vulnerabilities anywhere in a large client software stack can compromise confidentiality and integrity. We propose a new architecture for secure applications, Cloud Terminal, in which the only software running on the end host is a lightweight *secure thin terminal*, and most application logic is in a remote *cloud rendering engine*. The secure thin terminal has a very small TCB (23 KLOC) and no dependence on the untrusted OS, so it can be easily checked and remotely attested to. The terminal is also general-purpose: it simply supplies a secure display and input path to remote software. The cloud rendering engine runs an off-the-shelf application in a restricted VM hosted by the provider, but resource sharing between VMs lets one server support hundreds of users. We implement a secure thin terminal that runs on standard PC hardware and provides a responsive interface to applications like banking, email, and document editing. We also show that our cloud rendering engine can provide secure online banking for 5–10 cents per user per month.

1 Introduction

The ultimate motivation for much of computer security is protecting access to information: preventing unauthorized users from learning or altering sensitive data. However, current systems do a poor job of end-to-end information protection: applications are divided across multiple tiers, and each tier has many points of potential security vulnerability. One of the most vulnerable tiers is the software stack on end-user personal computers. To support a wide array of applications, operating systems and libraries contain millions of lines of code and evolve constantly. This complexity inevitably leads to vulnerabilities, and bugs anywhere in the stack can open the door for malware. For instance, if you use your PC for online banking, malware in your browser, your OS, or your drivers might log your keystrokes, steal your account information, or make transactions on your behalf. Common security mechanisms that run inside the OS can only offer limited protection, as the OS itself has a large attack surface. To provide the level of security needed by sensitive applications, we need to take the user-administered desktop OS out of the trusted computing base.

One frequently proposed approach to protect sensitive applications is to run them in their own virtual machines [12, 5]. This is a good first step, because virtualization offers strong isolation, but it is too heavyweight a solution to be secure and easy-to-use on end-user systems. Multiplexing hardware between several general-purpose OS VMs requires either a virtual machine monitor that itself runs on a general OS (e.g., VMware Workstation [1]), or a hypervisor that is almost as complex as a general OS (e.g., Xen [3]), so the client-side trusted computing base (TCB) is still too large to reliably secure. Managing multiple VMs also puts an administrative burden on users: installation tasks are multiplied, users may not know which VM to use for each task, and users must roll back any VMs that get compromised. Furthermore, it is difficult to introduce a general hypervisor, such as Xen, below an existing installation of a commodity OS. Security and usability could both be improved by keeping VM-style isolation, but changing how an application is decomposed between the client and server, so that the client can be both small and general-purpose.

Thus we propose a new architecture, which we call Cloud Terminal, for protecting applications from client-side security risks. We are motivated by the observation that the client side of many security-sensitive applications is predominantly concerned with providing an interface to information, rather than performing intensive computation or high-framerate animation. Thus we propose to move application-specific computations away from the hard-to-defend end-user platform. Instead, when accessing a sensitive application, we propose to use the end-user PC simply as a secure I/O path to access application logic in the cloud.

This architecture allows for a radically simpler client-side platform: the client-side *secure thin terminal* (STT) software only needs to render graphical data from a remote application and forward keyboard and mouse events to it. The STT isolates itself from the user's untrusted OS through a small hypervisor-like layer that we call a "microvisor." Because its scope is limited to running the STT, the microvisor is substantially simpler than a hypervisor: it does not need logic for time-sharing across multiple VMs, or even drivers for network and storage hardware (instead, it tunnels encrypted data through the untrusted OS). The simplicity of the STT makes it easier to assess the correctness of the software, and facilitates remote attestation. Leveraging a hardware root of trust, the client can prove to the server that it is running unmodified STT software, directly on the real CPU. The combination of strong isolation and attestation allows our STT to be installed and to securely function on any running system, even one infected with malware.

To complement the reduced client, we move application and rendering logic into a *cloud rendering engine* (CRE), which goes beyond cloud applications like web-based office suites. The cloud rendering engine executes an application all the way to producing a bitmap image to appear on the user's screen. It then sends that bitmap to the STT over an encrypted protocol. To provide strong isolation, the cloud rendering engine for each STT runs in a separate VM, but we show that because VMs can share state, one server can support hundreds of concurrent users. Hosting the application logic in a central location also allows providers to more easily manage software updates and protect information.

We argue that Cloud Terminal has the *minimal* client-side TCB needed for accessing remote applications from an untrusted system. Any system with this goal would need code for isolating itself from the OS, capturing user input, and displaying bitmaps on the screen, but these functions make up the *majority* of the code in our STT. Thus, Cloud Terminal achieves a unique and previously unmet sweet-spot between security and generality.

To summarize, our contributions in this paper are two-fold. First, we introduce the Cloud Terminal architecture, including the secure thin terminal and cloud rendering engine abstractions, as an effective new tool for building applications with strong information security. Second, we evaluate this architecture with realistic applications. We implement a secure thin terminal that runs on standard PC hardware, providing a small 23 KLOC TCB and TPM-based remote attestation. We build a cloud rendering engine from off-the-shelf software, and show that it supports hundreds of concurrent users per server. We then evaluate this infrastructure for applications including online banking, secure document editing, and email.

2 Overview

This section gives an overview of our architecture, including use cases, our threat model, a comparison with existing systems, and an overview of the design.

2.1 Use Cases

Cloud Terminal is designed for public and corporate applications that require high information security but not intensive computation or rendering. Many use cases satisfy these properties, including financial applications such as online banking, and communication applications that let corporate users access work data from their personal devices, such as a corporate email client.

In the public service scenario, users would install a single secure thin terminal that lets them access multiple services, such as banks and financial organizations, each hosting its own cloud rendering engine in its own datacenter. Financial services are a natural use case because they are a major target of fraud, end-users and institutions both have incentives to prevent attacks, and UI requirements are simple.

In the corporate scenario, employees would install a secure thin terminal distributed by their organizations to securely access applications like email, document viewing, and document editing from their personal computers. Accessing documents via Cloud Terminal, instead of downloading them to a personal device, significantly reduces the attack surface for data theft and malware.

2.2 Goals and Threat Model

Our aim is to design a solution that significantly improves the security of sensitive applications but requires minimal effort to adopt and use. Specifically, we seek to meet the following goals:

1. The solution should be installable on existing PCs alongside a potentially compromised commodity OS, without requiring the user to re-image her system.
2. The solution should not require trust in the host OS.
3. The solution should be able to attest its presence to both users and application providers, to protect against spoofing and phishing.
4. The system should support a wide range of sensitive applications.
5. The TCB of the system should be small.

We assume an adversary that controls the OS on the user's PC and can intercept all its network traffic, but does not have physical access to the machine to mount hardware attacks like cold-boot memory recovery [13]. We also assume that the attacker cannot reliably infer the user's input from sensors on the machine (e.g., by listening for keystroke timings on the microphone). Our goal is to prevent the attacker from viewing and modifying in-

Property	Red/Green VMs [18]	Per-app VMs [12, 29]	Browser OS (e.g. Chrome OS [7])	Virtual Desktops & Thin Clients	Flicker [20]	Cloud Terminal
Installable below an existing system	✘	✘	✘	✔	✔	✔
Remote attestation	✘	✘	✘	✘	✔	✔
Generic applications	✔	✔	✘	✔	✘	✔
Fine-grained isolation	✘	✔	✔	✘	✔	✔
No trust in host OS	✔	✔	✘	✘	✔	✔
User interface	arbitrary	arbitrary	browser only	arbitrary	✘	arbitrary
Management effort	medium	high	low	low	low	low
TCB size	>1MLOC	>1MLOC	>1MLOC	>1MLOC	250 LOC + app logic	23 KLOC

Table 1: Properties of Cloud Terminal compared to other security architectures.

teractions between the user and a secure application, and from logging into it as the user.

Cloud Terminal also protects against some social engineering attacks, such as users being tricked to run a false client, through remote attestation. In addition, it provides two defenses against phishing: a shared secret between the user and the secure thin terminal, in the form of a graphical theme for the terminal's UI, and the ability to use the user's TPM as a second, un-phishable authentication factor and detect logins from a new device. These mechanisms are similar to common mechanisms in web applications (e.g., SiteKey [32] and cookies for detecting logins from a new machine), with the important distinction that the secret image and the TPM private key *cannot* be retrieved by malware or by man-in-the-middle attacks. Nonetheless, we recognize that there are open problems in protecting users against social engineering and we do not aim to innovate on this front, as the problems are orthogonal to our focus on designing a well-isolated client.

Finally, because Cloud Terminal must coexist with the untrusted OS, it is not designed to prevent denial-of-service attacks from the untrusted OS: for instance, the untrusted OS could prevent the installation of the client in the first place, or block its network traffic.

2.3 Existing Approaches and Comparison

Although many architectures to improve the security of end-user systems have been proposed, it is challenging to simultaneously meet all the goals identified in the previous section. In this section, we compare several existing proposals (Table 1) and explain the elements of our approach that let it meet the goals.

One frequently proposed approach is to isolate sensitive applications using virtual machines, either by having a "red" VM for untrusted applications and a "green" VM for trusted ones [18], or one VM for each sensitive application [12, 29]. While VM-based solutions provide strong isolation, any hypervisor aiming for wide deployability needs a TCB comparable in size to an OS kernel, including drivers for all the the network and storage devices common on consumer PCs. Attacks targeting these components in popular hypervisors have already been demonstrated [17, 23, 41]. In addition, we need to include the applications inside the trusted VM (e.g., a web browser) in the TCB. Finally, VM-based systems are not readily installable below an existing commodity OS without requiring the user to reimage his machine, and they increase the administrative burden on users by requiring users to manage each VM separately.

In contrast, browser OSes like Chrome OS [7] limit their attack surface by disallowing binary applications and provide strong isolation between web applications. However, this means that they cannot run the user's existing legacy software or access non-web applications.

The approach closest to ours is virtual desktop infrastructure (VDI), where users access centrally managed virtual desktop VMs through thin client software [37]. While thin client systems allow organizations to centrally manage their desktops and remove infections, they still suffer from a fundamental limitation compared to Cloud Terminal, in that all of the user's applications run in the *same* VM and are not protected from each other. For example, a user's VM might become infected through a drive-by download from personal web browsing, which would then put all other applications at risk. Another limitation of current VDI systems is that the thin client software runs within the untrusted OS on the user's PC and is unprotected from malware on that machine.

Finally, remote attestation is challenging in VM-based approaches, browser OSes and VDI because of the wide range of software that can run on the client. For example, even if a system could prove that it is running a particular OS kernel and only a trusted set of binaries, the system could still be executing malware in the form of a shell script or a malicious browser extension. Thus, these approaches miss an opportunity to provide powerful security guarantees via TPM attestation. One example of a system that *does* support attestation is Flicker [20], which allows applications to run small pieces of application logic (PALs) in an isolated, attestable environment,

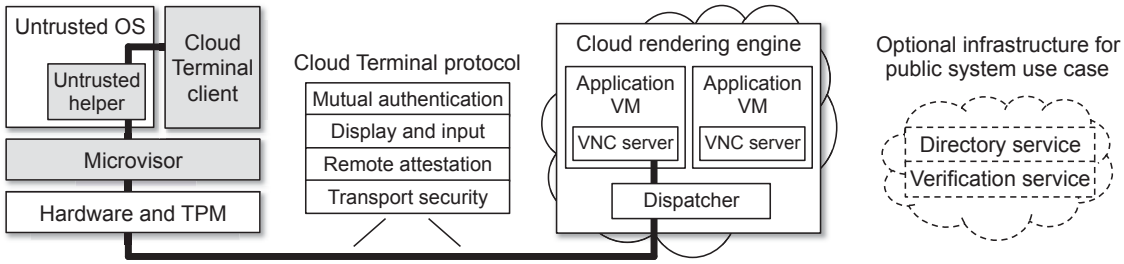


Figure 1: Cloud Terminal architecture, showing the secure thin terminal (left) and cloud rendering engine (right). The shaded areas make up the secure thin terminal.

for purposes such as key signing and password handling. However, executing a client for a remote application as a Flicker PAL would be difficult, because a PAL cannot interact with the user through the GUI, as Flicker suspends the untrusted OS during PAL execution.

Cloud Terminal achieves deployability on existing systems, support for general applications, remote attestation, and a small TCB through two design elements:

- **Small, general client:** Cloud Terminal accesses all sensitive applications through the same, simple component: a secure thin terminal capable of displaying arbitrary remote UIs. Thus, the user does not need to manage multiple VMs, and service providers can run their applications in their own datacenters under tight control. Unlike in virtual desktop systems, the sensitive applications are also isolated from each other (rather than running in the same VM), and the thin terminal is protected from the untrusted host OS.
- **Microvisor:** The secure thin terminal isolates itself from the OS through a hypervisor-like layer, but this "microvisor" is substantially smaller than a full hypervisor because it is not designed to run multiple VMs. For example, the microvisor accesses the network and storage devices through the untrusted OS (but it encrypts its data), leveraging the OS's existing drivers without having to trust them. Likewise, it does not need code for managing multiple VMs, or even for booting the untrusted OS; it can install itself below a running instance of the OS, and only needs to protect an area of memory from the OS.

This design lets Cloud Terminal achieve a sweet-spot between security, trusted code size, and generality: it can access a wide range of applications through a small, well-isolated, and remotely verifiable client.

2.4 Architecture

Figure 1 shows the overall Cloud Terminal architecture. Our approach centers on two abstractions: the *secure thin terminal* on the client and the *cloud rendering engine* on the server. We now describe these abstractions and show how they interact with other system components.

Secure Thin Terminal (STT). The STT is software that runs on a user's computer and provides secure access to a remote application, without requiring trust in any other software on the device. The STT temporarily takes over a general-purpose system, and turns it into a more limited but trustworthy device for accessing generic remote applications. The STT has the following features:

- The STT provides a common graphical terminal functionality that can be used by many applications.
- The STT isolates itself so that the untrusted system cannot access its data.
- The STT implementation is lightweight, making it easier to check its correctness.
- Using a hardware root of trust, the STT can remotely attest that it is running unmodified on real hardware.

The security of the STT comes from its simplicity: it focuses solely on providing an interface to applications running elsewhere. It provides this interface simply by relaying input events and remotely-rendered bitmaps. The STT co-exists with a pre-existing untrusted OS, but does not rely on the untrusted system for any security-critical functionality. Using hardware virtualization, the STT isolates itself from the untrusted OS: the OS never has unencrypted access to the STT's data, and cannot read input events or access the video memory when the STT is active. A hardware root of trust allows the STT to prove to remote parties that it has complete control of the machine, namely that its code is unmodified and that it has direct access to a real (non-emulated) CPU.

The STT consists of the *microvisor*, which provides isolation from the OS; the *Cloud Terminal client*, which communicates with the remote application and renders its display; and an untrusted user-space *helper* that tunnels encrypted data through the untrusted OS.

Cloud Rendering Engine (CRE). The CRE is STT's server-side counterpart. It has the following attributes:

- The CRE contains almost all the application functionality, to the point of producing bitmaps to display.
- The CRE runs an isolated instance of the application for each STT, in a separate virtual machine.

- CRE VMs run a minimal software stack needed to render the remote application, rather than allowing the user to install her own software.
- CREs are managed centrally by the application provider, facilitating administration and protection.

From some uses, such as document viewing, the CRE can be almost completely self-contained. For other uses, the CRE can provide access to a service on other machines, such as an existing web application, by running standard rendering software such as a web browser.

Our CRE implementation executes the application instance for each user session in a separate VM to provide strong isolation. These VMs run a stripped-down commodity desktop environment, connected to an unmodified VNC server that we proxy through a *dispatcher* for access by the appropriate STT. The main challenge in implementing the CRE is scalability: the system needs to be able to support a large enough number of users per server to be cost-effective. We employ aggressive sharing of disk and memory pages between application VMs to support several hundred concurrent users per CRE server.

Finally, although the CRE hosts a "cloud" of VMs, we expect it to run in a *private* provider-owned datacenter.

Cloud Terminal Protocol. The secure thin terminal and cloud rendering engine communicate over the network using the *Cloud Terminal protocol*. The Cloud Terminal protocol extends an existing framebuffer-level remote desktop protocol (VNC) by adding additional levels of security. Specifically, the Cloud Terminal protocol uses end-to-end encryption between the Cloud Terminal client in the STT and the CRE, performs remote attestation of the client, and provides mutual authentication between the user and application.

Public Infrastructure Services. If Cloud Terminal is deployed as a public service for accessing multiple applications via a single STT (e.g., financial websites that enroll in the system), rather than as a private service within a corporation, it must also host two infrastructure services. The *directory service* provides the client with a list of CREs to connect to (e.g., the CREs for various online banks). The *verification service* lets users check that they installed a genuine STT. Nonetheless, even in this use case, each application provider still hosts its own CRE on its own hardware. We describe this usage scenario in more detail below.

3 Secure Thin Terminal

The secure thin terminal has three components. The first is the *microvisor*, a small hypervisor-like layer providing isolation from the untrusted system and simple APIs for the following functionalities: keyboard and mouse input, video output, sealed storage, and networking. The microvisor can also attest (i.e., cryptographically prove) the integrity of its code to a remote third party through a *measurement* (a signed hash) from the TPM. The second component is the Cloud Terminal client, which runs within the microvisor. The third component is an untrusted user-space helper to which we delegate the handling of networking and storage API calls. We refer to this subset of the API calls as *untrusted API calls*. Through the helper, we can exploit the drivers present in the untrusted OS, and thus use any type of network controller (including WiFi and 3G cards), without having to take over and configure the network hardware.

Figure 2: Screenshot of the STT. The textured border around the browser (the strawberries) is a secret image configured by the user that will only be shown correctly by the genuine STT.

The secure thin terminal can be installed on a system at any time, even if malware is present. Once the STT has been installed, the user can bring up the Cloud Terminal client by pressing a designated secure attention key (e.g., Ctrl-F12). While the client is running, the untrusted OS and its applications continue to run, but blindly: we prevent them from seeing the user's inputs and the contents of video memory. Figure 2 shows the STT's UI.

3.1 Microvisor

The simplicity and small code size of the microvisor come from its limited scope (e.g., no network and storage drivers) and from leveraging hardware virtualization support (Intel VT [22, 14]). For attestation, we employ Intel's trusted execution extension, TXT [15]. The combination of hardware-supported virtualization and trusted execution allows us to establish a tamper-proof, measured execution environment.

When the Cloud Terminal client is not running, the microvisor is responsible only for isolating itself from the untrusted OS and applications. Specifically, the microvisor makes its address space inaccessible by preventing the untrusted system and the I/O devices from accessing those ranges of physical memory. Additionally, the

microvisor intercepts keystrokes to detect when the user requests to bring up the Cloud Terminal client. It does so by trapping reads made by the untrusted system to the PS/2 port, emulating the read, and updating the state of the untrusted system accordingly, as if the read had been performed directly. As we shall describe, we use a similar approach to capture the user's input securely when the Cloud Terminal client is running. For video output, the microvisor maps the video memory into its virtual address space and then renders images on the screen by directly writing to the memory.[1]

In our current implementation, the microvisor is installed after the untrusted OS has started running; nevertheless, it assumes complete control of the system, in a similar manner to malicious hypervisors [9, 28]. This approach does not require any prior setup of the system.

To securely install the microvisor, we use code from the Flicker secure execution infrastructure [20], which in turn uses Intel TXT [15]. This primitive allows us to establish a dynamic root of trust for measurement and attestation: it ensures that the code for performing the installation of the microvisor and the code of the microvisor cannot be tampered with (the code is executed atomically and it is stored in a region of memory that is completely isolated, even from hardware devices) and stores a measurement (hash) of this code in the TPM. The Flicker-attested code also generates a key pair whose private component is kept only by the microvisor, linking future communications to the attested execution. The private key is kept in volatile RAM whose contents will be lost if the untrusted OS forces a reboot, so no other code can masquerade as the microvisor.

We start the installation by saving the current state of the untrusted system so that we can later resume it as a "guest" of the microvisor. Then, we invoke the `senter` instruction and run the code performing the necessary steps to enable the microvisor. Finally, we resume the execution of the untrusted system from where it was interrupted. From this point on, the microvisor has full control of the system and it is responsible for isolating itself from the untrusted OS. As described in Section 5, the measurement stored in the TPM is used to prove to the cloud rendering engine that the client is genuine.

3.2 Cloud Terminal Client

The Cloud Terminal client is essentially a process that runs in the context of the microvisor and interacts only through the microvisor's API. The client starts by making a backup of the contents of video memory and ends with restoring these contents and redrawing the screen. The rest of the execution implements the Cloud Terminal protocol. Untrusted API calls and API calls to get user input block the client and resume the execution of the untrusted OS (without access to the video memory, as we describe later). The client is then resumed when a blocking call returns. To ensure that no sensitive data can be accessed outside the Cloud Terminal client, before and after an untrusted API call the client respectively encrypts the input arguments and decrypts the output arguments. Data transmitted over the network are encrypted using the shared session key established during the initial stage of the protocol. Data stored on disk are encrypted with a symmetric key that we store persistently in the TPM using *sealed storage* [20], ensuring that the key can only be retrieved by a genuine STT.

3.3 Securing the Execution of the Client

During the execution of the client, the microvisor transparently dispatches untrusted API calls to the untrusted helper. Since the untrusted system must continue to run while the client is running, we have to ensure that an attacker cannot see the video output of the secure thin terminal and that he cannot sniff inputs coming from the keyboard and the mouse.

To read keystrokes and mouse events from the client, we intercept and emulate reads from the PS/2 port, but do not deliver sensitive events to the untrusted system. To prevent an attacker from seeing what is being rendered on the screen, we hijack the virtual mapping of the video memory in the untrusted system. Specifically, we configure the MMU to redirect accesses to the memory region mapping the video memory to the region that we use as a backup. Then, we ensure that the untrusted system cannot map the real video memory. When the Cloud Terminal client is terminated, we restore the original mapping, content, and permissions of the video memory.

3.4 Untrusted User-Space Helper

The helper program runs in user-space in the untrusted system and is very simple: it provides basic networking and storage capabilities but is aware of the data it manages. Since sensitive data do not leave the microvisor unencrypted, a compromised helper cannot violate data confidentiality or integrity.

The helper and microvisor share a memory region. The microvisor makes requests by writing to the region, and the helper signals completion using a hypercall (the `vmcall` instruction, similar to a system call). The helper using polling, with a frequency set by the microvisor, to avoid the need to modify the untrusted OS.

4 Cloud Rendering Engine

The cloud rendering engine (CRE) runs instances of an application to render bitmaps displayed by the STT. It

[1] We currently do not support USB keyboards and mice, though we plan to do so in the future. We also require the host OS to be configured with hardware graphics acceleration disabled so that we can correctly manipulate its graphics state.

consists of a *dispatcher* that accepts connections from STTs and one *application VM* for each session. The software to run in each VM is chosen by the service provider, but may include a web browser to render an existing web application, a word processor for secure document editing, or in-house enterprise applications.[2]

When a client connects to the CRE, the dispatcher assigns a VM cloned from a base image to run its application instance. We run a remote desktop server, such as a VNC server, inside each VM to render the application's UI to bitmaps, and send these to the client.

4.1 CRE Scalability

The main challenge in designing the CRE is scalability: for cost efficiency, we wish to support hundreds of application VMs per CRE server. We leverage two properties of the CRE workload to achieve this. First, interactive applications spend most of their time waiting for input, allowing for statistical multiplexing of CPU and network resources. Second, because the VMs all run the same software, they can share a high fraction of memory [38, 40, 39]. Although the specific sharing mechanisms we use are not new, our contribution is the observation that these techniques work especially well for a CRE workload, providing far higher savings than in traditional server consolidation, and thus making CREs cost-effective.

By leveraging these features of the workload, we can support several hundred concurrent users running rich desktop applications, such as Firefox, on a single commodity server at a cost of few cents per user-hour (see Section 6.3). We describe the key optimizations below.

Memory sharing. We coalesce identical memory pages from the application VMs into a single page with copy-on-write behavior. We found that this can reduce the footprint of each guest VM by 38% to 61%.

Disk sharing. Each VM uses a copy-on-write disk image based on a single master image. This minimizes the number of extra blocks needed per VM and keeps the base image in the server's buffer cache for fast access.

Stripped-down OS. The guest VMs run only the software needed to support the desired application.

Reduced timer interrupts. We lowered the CPU usage of our guests substantially by configuring them to run a "tickless" Linux kernel, which uses ACPI to set timers instead of needing a periodic 100 Hz interrupt [31]. This change reduced the CPU usage of each idle guest from 10% of a core to less than 1%.

[2] We used VMs for isolation instead of lighter-weight containers to show that high scalability can be achieved even with strong isolation.

4.2 CRE Security

The CRE provides significantly more protection to the application than an operating system on the user's machine because it accepts only sessions from attested STTs, receives only keyboard and mouse input from these clients, runs up-to-date software chosen only by the service provider, and does not expose guest VMs to the Internet. Nonetheless, there is a risk that users running sessions on the same CRE server could interfere with each other. We mitigate this risk in three ways:

Network isolation. The client VMs run on separate virtual networks behind a NAT, so that they cannot send traffic to each other. We also restrict their access to the external network using a firewall, so that they can only communicate with servers necessary for their applications (e.g., a web server for the banking use case).

Resource isolation. We use standard VM isolation features to cap guests' memory, disk, network and CPU use.

Restricted user environment. The applications in each guest VM run on a Linux user account with minimal privileges, inside a desktop environment that does not let the user launch any other applications.

In the end it may still be possible for a malicious user to subvert the software running inside a CRE VM, such as a web browser or the Linux kernel, but the CRE should have no more authority that the user it is assigned to. Even if the user gained full control of her VM, she would not be able to access other users' VMs over the network. Cross-VM information leakage [27] may be a concern, but we believe that the information gain from such leakage is limited, especially because of the high number of VMs running on each CRE server.

5 Setup and Session Protocols

We now discuss how users install Cloud Terminal (Section 5.1) and how the STT and the CRE communicate (Section 5.2). We focus on the first use case in Section 2.1, where the user installs a single STT to access multiple public applications, such as banking sites. The private deployment case is similar. We include some extensions to the protocol, such as multi-factor authentication and copy-and-paste support, in an extended version [11].

5.1 Cloud Terminal Installation

The STT installation process seeks to achieve two goals: (1) certifying to the user that she is installing a genuine STT and (2) establishing a shared secret between the STT and the user, so that the user can check whether she is accessing the real STT in the future.

For the first goal, we include a *verification service* as part of the public Cloud Terminal infrastructure that communicates with users through a secondary channel (a phone) to let them verify their STT. When the STT

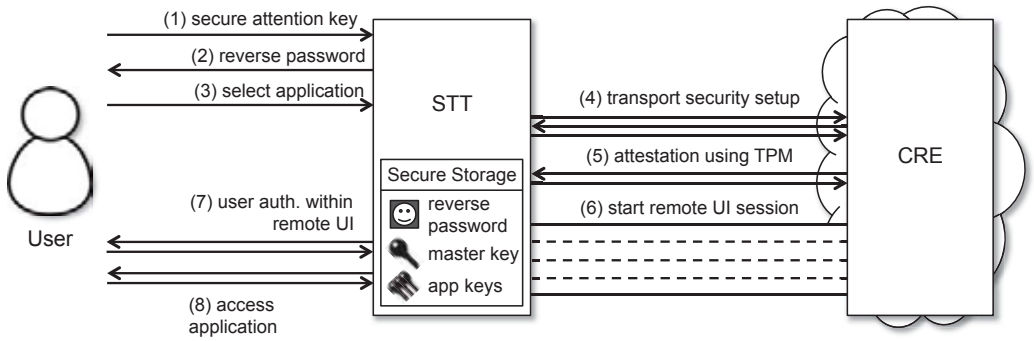


Figure 3: Process for a Cloud Terminal session. The user presses a key (1) to open an STT UI authenticated by the reverse password (2). She then picks an application (3). We use an application key to set up a secure transport session (4), attest the validity of the STT using the TPM (5), then set up a remote UI session (6). The user logs in through this UI (7) to access the application (8).

starts for the first time, it connects to the verification service, attests its presence on the machine using the TPM, and receives a random nonce value from the verification service which it displays to the user. The user then calls the verification service at a well-known number, enters the nonce, and is told whether the nonce corresponds to a correctly attested STT. Due to space constraints, we discuss this process in more detail in [11].[3]

For the second goal, we have the user select a background image to be shown on the STT's UI, which we call a *reverse password* because its role is to authenticate the software to the user rather than the user to the software [32]. The STT keeps this image in sealed storage, so that malware cannot obtain it and phishers cannot guess it. However, because users have been shown prone to social engineering attacks against similar authentication schemes in web applications [30], the reverse password is *not* our only defense against phishing; as we will discuss in Section 5.2, applications can also use the TPM's private key can as a second authentication factor to detect logins that are not from the user's device.

5.2 Session Protocol

Once the user has installed the STT, she can launch its UI at any time by pressing a special key intercepted by the microvisor. This UI displays a list of available applications that the STT obtains from a *directory service* managed by the Cloud Terminal provider. We expect providers to require substantial verification before adding applications to this directory (e.g., similar to Extended Validation SSL certificates). The STT connects to the directory service securely using a *master public key* for the service stored in the STT binary, and gets back an *application public key* for each application.

Once the user selects an application, the STT opens a session to it through the Cloud Terminal protocol. This is a simplified TLS-like protocol that also uses the TPM to attest that the user is running a valid STT, and shows a remote UI through a subset of VNC. Through the remote UI, the user can log into the service as usual (e.g., with a password). Overall, session setup consists of the following steps, also shown in Figure 3:

1. The user presses a special key to bring up the STT.
2. The user checks that this screen displays the correct reverse password image to authenticate the STT.
3. The user selects the application to access.
4. The STT connects to a CRE for the application using its known DNS name, by tunneling network access through the helper in the OS. It uses the application's public key to authenticate the server and sets up an encrypted connection through a subset of TLS [10].[4]
5. The CRE verifies that it is talking to a valid STT through TPM attestation: it sends a nonce to the STT and gets back a combination of the nonce and a hash of the executing code, signed by the TPM. The STT also sends the CRE a hash of the key of the application it connected to, also signed by the TPM, so that a malicious CRE cannot proxy a session to another CRE by forwarding its nonce.
6. The CRE renders the application's UI through a framebuffer protocol similar to VNC.[5]

[3] One challenge with this process is knowing that the STT that contacted the verification service is running on the *user's* machine. Parno observed that any verification process with current TPMs is vulnerable to a "cuckoo attack" where an adversary gives the user a trojan horse to install but runs a real TPM on a different machine and relays its UI to the user [24]. He proposes several solutions, the most attractive of which is to have PC vendors engrave a hash of the TPM's public key on each device. The user could then confirm this hash to the verifier by sending a verified-selected subset in an SMS message. Another measure that reduces the viability of this attack is to only accept TPM keys signed by a hardware manufacturer and to only accept each key once, so that the adversary must acquire a new TPM chip for each attack.

[4] We used RSA for key exchange, 128-bit AES CBC for symmetric encryption, and HMAC-SHA256 for the MAC.

[5] We chose a subset of the RFB protocol [26] used by VNC. RFB

7. Within this remote UI, the application displays an interface of its choice to authenticate the user.

In addition to letting the CRE verify that it is communicating with an unmodified STT running on the physical CPU, the TPM attestation step can also be used by applications to defend against phishing. The key idea is that even if an attacker obtains the user's account name and password, she cannot obtain the private key of the TPM on the user's device: even the user herself does not know it! Thus, the application can remember the TPM public key it saw for each user and treat sessions that employ a different TPM as suspicious, asking the user to confirm that they are using a new device through a secondary channel like SMS. Many current web applications similarly detect logins from a new device using browser cookies, but the TPM private key has the advantage that it cannot be read by malware, unlike cookies.

6 Implementation and Evaluation

To evaluate the Cloud Terminal architecture, we have implemented a secure thin terminal that runs on standard PCs and a cloud rendering engine based on off-the-shelf software. This section describes our implementation (§6.1) and four applications: banking, secure document viewing, document editing, and email (§6.2). We then evaluate the system's performance (§6.3) and estimate the cost for a provider to run Cloud Terminal (§6.4).

6.1 Implementation

Secure Thin Terminal. We have implemented all the client components of our architecture. All components are available for Linux, but the attestation in Linux is currently only partially secured (i.e., not all the code of the system is measured). For simplicity, the address space of the Cloud Terminal client is currently not isolated from the address space of the microvisor, though this could be achieved by running the client in an unprivileged ring. We have not yet implemented support to protect the STT from malicious device DMAs (using VT-d) or from malicious SMI handlers. Also for simplicity, our implementation of the RFB protocol supports only raw image encoding and does not perform any compression.

Our trusted computing base consists of 20.5K lines of C and 1.4K lines of assembly, distributed as follows. The microvisor consists of 7K lines of C and 0.7K lines of assembly. The client consists of 3K lines of C (excluding two large array initializers for a bitmap font and the system logo). Our cryptographic routines are from the PolarSSL library [25] (we borrowed 5.5K lines of C) and our TPM based attestation code is based on Flicker (5K lines of C and 0.7K lines of assembly). (Measurements are all non-comment-non-blank physical lines.)

explicitly allows clients to support a subset of protocol options, which allows our cloud rendering engines to use an off-the-shelf VNC server.

Operation	Time (msec)
PCR Extend of Flicker (PCR-18)	3.43
Key Generation	24.25

Table 2: Time in milliseconds required for some of the operations in the attestation protocol of the Thin Client over averaged over 100 trials. All of the operations occur within the TCB.

We tested the Linux implementation of the secure thin terminal on a Lenovo W510 laptop. This laptop has a PS/2 keyboard and mouse, supports virtualization technology and trusted execution, and includes an STMicroeletronics TPM.

Table 2 depicts some of the main operations that are necessary in performing the remote attestation. The Thin Client is responsible for performing these operations within its TCB.

Cloud Rendering Engine. We implemented the CRE on Linux, using KVM [16] as our virtual machine manager. We leverage Linux's Kernel Samepage Merging (KSM) daemon [2] to share identical memory pages. We keep a pool of fresh VMs for new sessions to hide startup times.

For our guest OS, we used a Debian GNU/Linux 6 installation with minimal packages to run the desired application (e.g., Firefox), a simple window manager (XFWM) and a VNC server. Our experiments used a resolution of 800x600 pixels and 8 bits-per-pixel of color depth, which we found sufficient for a single application (see Figure 2). We also configured the window manager to prevent the user from launching other programs.

6.2 Applications

We built four applications over Cloud Terminal: online banking, document viewing, document editing, and secure email. Very little effort was required to implement these applications: two give access to existing web applications, while the other two use existing Linux software.

Online banking. We built a CRE for the Wells Fargo website, which displays the existing site using Firefox. We run Firefox in kiosk mode: the page fills the entire desktop and the user cannot execute other applications. We block off-site resources and links with an HTTP proxy. The bank can easily configure a whitelist for this proxy.

Document viewing. As an example of a corporate application for Cloud Terminal, we built a CRE for viewing sensitive PDF documents via Evince, a Linux PDF viewer. We currently show a local document in each VM, though it would be easy to give the VMs access to the company's network file system after users authenticate.

Document editing. In addition to viewing documents, we let users edit them through a CRE that runs AbiWord, a word processor compatible with Microsoft Word.

Application	Activity	Baseline (ms)	STT (ms) with # of concurrent clients =						Network Usage (bytes)	
			1	50	100	150	200	300	inbound	outbound
Doc. editing	Launch app.	2,844	1,732	2,000	2,033	2,208	2,441	2,553	487,047	3,888
	Type a key	30	50	52	51	53	50	54	1,607	346
	Move the mouse	32	47	48	53	49	59	51	480	138
Doc. viewing	Launch app.	1,699	1,817	1,947	2,066	2,093	2,147	2,493	483,219	2,040
	Scroll a page	114	1,175	1,208	1,281	1,270	1,380	1,704	352,358	5,497
Online banking	Launch app.	6,911	1,581	2,047	2,232	2,319	2,563	____	490,149	4,680
	Browse a new page	1,183	2,302	2,516	2,573	2,610	2,661		415,732	10,939
Secure email	Launch app.	6,936	1,865	1,992	2,054	2,254	____	____	488,367	3,954
	Display an e-mail	992	1,976	2,160	2,106	2,254			318,300	8,416

Table 3: End-to-end UI latency and network traffic at the STT for various actions. Measurements are averaged over 30 runs.

Secure email. As an example of a richer web application, we also used Firefox to provide access to Gmail. For example, an organization might implement this service to let employees access sensitive email from their home PCs. Although Gmail is more CPU-intensive than our other applications, we could still support more than a hundred concurrent sessions on one server.

6.3 Performance Evaluation

We sought to answer two questions with our performance evaluation. First, how responsive is the STT as a means for accessing remote applications? And second, how far can a CRE scale while providing a good user experience? We omit an evaluation of time to launch the STT and overhead imposed by the microvisor on the user's OS, because these overheads are negligible and have been measured in detail for similar primitives [19].

To answer these questions, we ran a CRE on a 16-core server with 2.0 GHz Opteron processors and 64 GB RAM. We then connected up to 300 emulated clients to it to generate load. These clients replayed packet traces from our STT implementation to loop a 3–5 minute recorded UI session.[6] Finally, we connected a "probe" STT instance running on a laptop that we interacted with manually to measure UI responsiveness. This client experienced a 23 ms network latency from the Berkeley campus network to our CRE hosting provider in Seattle.

In summary, our results show that each of our applications scaled to 150–300 simultaneous sessions on a single server while providing a responsive experience. Furthermore, the network bandwidth per session was well within the means of current ISPs. The main limiting resource on the CRE server was the CPU. We stopped scaling each application when its CPU load reached 90%.

6.3.1 Qualitative Usability

Most importantly from a usability perspective, the system felt usable qualitatively. We were able to type paragraphs of text unhindered by the refresh speed (typing latencies were similar to SSH), and to comfortably navigate through the applications. The slowest action was scrolling the page, which we have not yet optimized.

6.3.2 Client-side Metrics

To quantitatively demonstrate the usability of STT, we measure the end-to-end UI latency at the client for various user activities, such as typing a key, navigating to a new page, and loading an application. End-to-end UI latency for a user activity is the amount of time taken from when the user begins an action and until the system finishes rendering the result of that action. Table 3 presents the average UI latency on an STT client when running with different levels of CRE server load, as well as the amount of bandwidth used. We compare these measurements against a baseline where we run the application locally. For startup time, this baseline is after a reboot, so it includes the time to load the program from disk.

The results in Table 3 show that the latency introduced by the STT is low for most activities. For keystrokes, Cloud Terminal adds up to 24 ms of latency even when the CRE is serving 300 concurrent users, bringing the total latency up to 54 ms. This is substantially lower than the average inter-keystroke timing of 100 ms [33]. For activities that refresh a large part of the screen as opposed to displaying one new character, like navigating to a new page, we see between 1.0 and 1.6 seconds of extra latency. Much of this is due to unoptimized rendering and image compression that we believe can be improved without substantial effort (e.g., we can speed up scrolling by sending a command to translate part of the image).

Interestingly, launching an application for the first time was often *faster* via Cloud Terminal than locally, because the application was pre-loaded in a CRE VM.

6.3.3 Server-side Metrics

We report the load on the CRE server running each application with varying numbers of clients in Table 4. We see several interesting trends. First, CPU was always the

[6] The sessions were: opening an account on Wells Fargo, reading a PDF, editing a Word document, and reading/writing emails.

Application	Document editing			Document viewing			Online banking			Secure email		
and # clients	100	200	300	100	200	300	100	150	200	50	100	150
CPU load	27%	54%	84%	34%	66%	96%	37%	60%	81%	28%	57%	84%
Mem. GB	8.4	15.1	22.1	8.4	15.7	25.2	25.5	38.3	47.6	15.1	26.4	38.1
KB/s in	367	708	1067	286	553	769	1059	1607	1931	251	392	591
KB/s out	2054	4360	8951	2993	5562	7709	2568	3527	4395	1802	3468	4888

Table 4: CRE load for each application with various numbers of concurrent clients.

limiting resource for our server configuration. This includes both the cost of encrypting traffic and of running the application itself. Second, both CPU and memory usage were 2–4× higher for the browser-based applications (Wells Fargo and Gmail) than the standalone Linux applications. Gmail was especially CPU-intensive due to its heavy use of JavaScript. Nonetheless, even these applications were able to scale to 150 and 200 sessions on a single CRE server, while running in Firefox 3. Finally, bandwidth usage stayed below 9 MB/s in total, or, on average, at most 32 KB/s per session, which is within the capabilities of both home ISPs and hosting providers.

6.4 Cost Analysis

We estimate the per-user cost of running a Cloud Terminal service based on our measurements and on hosting prices at one provider (CariNet [6]). CariNet offers a 12-core server with 40 GB RAM and unmetered 100 Mbps connectivity for $1010/month. Assuming that this server can run 3/4 of our 16-core load, it can host 82,000 to 164,000 user-hours per month. This makes the overall cost between 1.2 and 2.5 cents per user-hour. For an online banking service, the average user is unlikely to log in for more than 2 hours per month (based on an informal poll of our group), making the cost up to 5 cents per user per month. For a corporate application, the cost is at most $3 per employee per month even if the employees use the service 8 hours per day.

7 Related Work

Section 2.3 has already compared Cloud Terminal to several previous approaches for secure access to applications, including systems that isolate applications using VMs [18, 12, 29], browser OSes [7], thin clients [37], and Flicker [20]. Cloud Terminal distinguishes itself from these systems by simultaneously providing five properties: installability under an existing (potentially compromised) OS, remote attestation, support for general applications, isolation across applications, and a small enough TCB to make verification feasible (23 KLOC). The main insight allowing this is the choice of a much simpler, but still general client: a thin terminal. This client is simple enough to remotely attest yet capable of displaying arbitrary UIs.

Several other related projects include Tahoma [8], a browser OS that isolates web applications using virtualization, and IBOS [35], a microkernel-based browser OS. Both approaches reduce the trusted code base on the client (although, unlike Cloud Terminal, they must include drivers for network and storage devices in the TCB), but they are limited to protecting web applications and they do not provide remote attestation. Proxos [34] partitions a system call interface so that a "private application" has certain requests serviced a VMM-isolated "private OS" and others by a commodity OS, similarly to how the STT interacts with some devices directly and others via the untrusted helper. One application of Proxos is to protect a web browser, but for this application the TCB includes both Xen and an X server.

Remote attestation has been explored by several projects, including Tboot, which can perform a measured and verified bootstrap of an OS or of a hypervisor [36], and TrustVisor, a hypervisor that relies on hardware attestation to ensure code integrity and secrecy for selected portions of an application [19]. In contrast to these systems, Cloud Terminal can run an entire, interactive UI session in an attestable execution environment.

Several projects have looked at small-TCB approaches to secure sensitive inputs to online services, such as credit card numbers and passwords, in the presence of malware. Bumpy [21] allows users to type a secure attention sequence to encrypt input for a web service, but takes a hardware approach (using an encrypting keyboard) in contrast to our software approach. The Trusted Input Proxy (TIP) [4] uses a hypervisor and a separate VM to pop up dialog boxes for sensitive input whose responses are injected in a TLS stream. However, TIP does not provide confidentiality for the whole UI session—for example, malware can still see the account statements sent back by a bank or the contents of a document being edited. In contrast, the STT hides an entire UI session.

In concurrent work, Zhou et al. [42] describe how to protect trusted I/O paths from device-level attacks such as overlapping memory-mapped I/O and spoofed interrupts. The STT relies on such trusted paths to the keyboard and display, and could benefit from many of the techniques they propose.

Finally, the cloud rendering engine can be seen as an extreme form of Software as a Service (SaaS). By running almost all of the application logic centrally—everything except the I/O path—the CRE offers two advantages to application providers: it makes it easier to

update the application (without requiring users to download a binary client in order to benefit from new features or security fixes to most of the code), and it allows for a simpler, more secure, and remotely verifiable client.

8 Conclusion

We presented Cloud Terminal, a new architecture for secure applications built around two primitives: a small *secure thin terminal* (STT) on the client and a *cloud rendering engine* (CRE) that contains almost all the application logic. The STT can be installed under a running operating system on standard PC hardware, even on compromised machines, and can be verified remotely. It achieves a sweet-spot between security, trusted code size, and generality by implementing a remote display protocol that can render arbitrary applications. The CRE runs applications in a provider-managed cloud and can scale to hundreds of sessions per machine. We have shown that Cloud Terminal is implementable on standard hardware and can provide secure access to a variety of applications at a low cost of 1.2 to 2.5 cents per user-hour.

Acknowledgments

Our shepherd Jon Howell suggested a change to the verification protocol to reduce assumptions about the phone system. The work described here has been supported by the NSF under awards CCF-0424422, 0842695, 0831501, and CNS-0831535, the AFOSR under MURI awards FA9550-08-1-0352 and FA9550-09-1-0539, Intel through the ISTC for Secure Computing, a Google PhD fellowship, and the NSERC (Canada). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funders.

References

[1] O. Agesen, A. Garthwaite, J. Sheldon, and P. Subrahmanyam. The evolution of an x86 virtual machine monitor. *SIGOPS Oper. Syst. Rev.*, 44:3–18, Dec. 2010.

[2] A. Arcangeli, I. Eidus, and C. Wright. Increasing memory density by using KSM. In *Linux Symposium*, pages 19–28, July 2009.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

[4] K. Borders and A. Prakash. Securing network input via a trusted input proxy. In *HotSec*, 2007.

[5] K. Borders, E. V. Weele, B. Lau, and A. Prakash. Protecting confidential data on personal computers with storage capsules. In *USENIX Security Symposium*, Aug. 2009.

[6] Compare dedicated servers from CariNet. `https://www.cari.net/compare-all-servers`.

[7] Google Chrome OS. `www.chromium.org/chromium-os`.

[8] R. Cox, J. Hansen, S. Gribble, and H. Levy. A safety-oriented platform for web applications. In *IEEE Security & Privacy*, 2006.

[9] D. Dai Zovi. Hardware Virtualization Based Rootkits. In *Black Hat USA*, 2006.

[10] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol — version 1.2. IETF RFC 5246, Aug. 2008.

[11] Full-length version of present paper. `http://bitblaze.cs.berkeley.edu/cloudterm.html`.

[12] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *SOSP*, 2003.

[13] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security Symposium*, 2008.

[14] Intel Corporation. Intel Virtualization Tech. for Directed I/O.

[15] Intel Corporation. Intel Trusted Execution Technology (TXT).

[16] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux virtual machine monitor. In *Linux Symposium*, 2007.

[17] K. Kortchinsky. Cloudburst: Hacking 3D (and breaking out of VMware). In *Black Hat USA*, 2009.

[18] B. Lampson. Accountability and Freedom. (Presentation), 2005.

[19] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. D. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy*, 2010.

[20] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *ACM EuroSys*, Apr. 2008.

[21] J. M. McCune, A. Perrig, and M. K. Reiter. Safe passage for passwords and other sensitive data. In *NDSS*, 2009.

[22] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Tech. Journal*, 10(3):167–177, Aug. 2006.

[23] T. Ormandy. An empirical study into the security exposure to hosts of hostile virtualized environments. In *CanSecWest*, 2007.

[24] B. Parno. Bootstrapping trust in a "trusted" platform. In *HotSec*, 2008.

[25] PolarSSL: Small cryptographic library. `www.polarssl.org`.

[26] T. Richardson. The RFB Protocol (ver. 3.8), Nov. 2010. `www.realvnc.com/docs/rfbproto.pdf`.

[27] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS*, 2009.

[28] J. Rutkowska. Subverting Vista Kernel For Fun And Profit. Black Hat USA, 2006.

[29] J. Rutkowska and R. Wojtczuk. QubesOS. `www.qubes-os.org`.

[30] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The emperor's new security indicators. In *IEEE Sec. & Privacy*, 2007.

[31] S. Siddha, V. Pallipadi, and A. Van De Ven. Getting maximum mileage out of tickless. In *2007 Linux Symposium*, 2007.

[32] SiteKey – Bank of America. `www.bankofamerica.com/privacy/index.cfm?template=sitekey`.

[33] D. Song, D. Wagner, and X. Tian. Timing analysis of keystrokes and timing attacks on SSH. In *USENIX Security Symposium*, pages 337–352, Washington, D.C., USA, Aug. 2001.

[34] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *OSDI*, 2006.

[35] S. Tang, H. Mai, and S. T. King. Trust and protection in the Illinois browser operating system. In *OSDI*, 2010.

[36] Trusted Boot. `http://tboot.sf.net/`.

[37] VMware Inc. Virtual Desktop Infrastructure.

[38] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *SOSP*, 2005.

[39] C. A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36:181–194, Dec. 2002.

[40] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the Denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36:195–209, December 2002.

[41] R. Wojtczuk. Subverting the Xen hypervisor. In *Black Hat USA*, 2008.

[42] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building verifiable trusted path on commodity x86 computers. In *IEEE Symposium on Security and Privacy*, 2012.

Mosh: An Interactive Remote Shell for Mobile Clients

Keith Winstein and Hari Balakrishnan
M.I.T. Computer Science and Artificial Intelligence Laboratory, Cambridge, Mass.
{keithw,hari}@mit.edu

Abstract

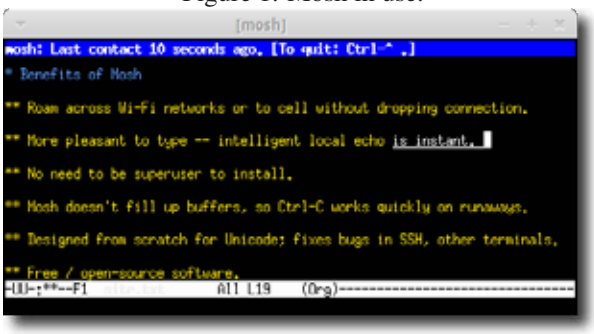
Mosh (mobile shell) is a remote terminal application that supports intermittent connectivity, allows roaming, and speculatively and safely echoes user keystrokes for better interactive response over high-latency paths. Mosh is built on the State Synchronization Protocol (SSP), a new UDP-based protocol that securely synchronizes client and server state, even across changes of the client's IP address. Mosh uses SSP to synchronize a character-cell terminal emulator, maintaining terminal state at both client and server to predictively echo keystrokes. Our evaluation analyzed keystroke traces from six different users covering a period of 40 hours of real-world usage. Mosh was able to immediately display the effects of 70% of the user keystrokes. Over a commercial EV-DO (3G) network, median keystroke response latency with Mosh was less than 5 ms, compared with 503 ms for SSH. Mosh is free software, available from `http://mosh.mit.edu`. It was downloaded more than 15,000 times in the first week of its release.

1 Introduction

Remote terminal applications are almost as old as packet-switched data networks. The most popular such application today is the Secure Shell (SSH) [9], which runs inside a terminal emulator. Unfortunately, SSH has two major weaknesses that make it unsuitable for mobile use. First, because it runs over TCP, SSH does not support roaming among IP addresses, or cope with intermittent connectivity while data is pending, and is almost unusable over marginal paths with non-trivial packet loss. Second, SSH operates strictly in character-at-a-time mode, with all echoes and line editing performed by the remote host. On today's commercial EV-DO and UMTS (3G) mobile networks, round-trip latency is typically in the hundreds of milliseconds when unloaded, and on both 3G and LTE networks, delays reach several seconds when buffers are filled by a concurrent bulk transfer. Such delays often make SSH painful for interactive use on mobile devices.

This paper describes a solution to both problems. We have built **Mosh**, the mobile shell, a remote terminal application that supports IP roaming, intermittent connectivity, and marginal network connections. Mosh performs predictive client-side echoing and line editing without any change to server software, and without regard to which application is running. Mosh makes remote servers feel more like the local computer, because most keystrokes are reflected immediately on the user's display—even in full-screen programs like a text editor or mail reader.

Figure 1: Mosh in use.



These features are possible because Mosh operates at a different layer from SSH. While SSH securely conveys an octet-stream over the network and then hands it off to a separate client-side terminal emulator to be interpreted and rendered in cells on the screen, Mosh contains a server-side terminal emulator and uses a new protocol to synchronize terminal screen states over the network, using the principle of application-layer framing [3].

Because both the server and client maintain an image of the screen state, Mosh can support intermittent connectivity and local editing, and can adjust its network traffic to avoid filling network buffers on slow links. As a result, unlike in SSH, in Mosh "Control-C" always works to cease output from a runaway process within an RTT.

Mosh's design makes two principal contributions:

1. **State Synchronization Protocol:** A new secure object synchronization protocol on top of UDP to synchronize abstract state objects in the presence of roaming, intermittent connectivity, and marginal networks (§2).
2. **Speculation:** Mosh maintains the screen state at both the server and client and uses the above protocol to synchronize them (§3). The client makes guesses about the effect each new keystroke will have on the screen, and when confident renders the effects immediately. The client verifies its predictions and can repair the screen state if necessary.

We have implemented Mosh in C++ and have experimented across various networks and across disconnec-

tions (§4). Mosh is free software, distributed with a variety of operating systems and at `http://mosh.mit.edu`. Mosh was downloaded more than 15,000 times in its first week of release in April 2012. An example of Mosh's interface is shown in Figure 1.

2 State Synchronization Protocol

Mosh works to convey the most recent state of the screen from server to client at a "frame rate" chosen based on network conditions. This allows the server to avoid filling up network buffers, because it does not need to send every octet generated by the application. (The reverse direction has less flexibility because the client must send every keystroke to the server.)

Supporting this is SSP, a lightweight secure datagram protocol to synchronize the state of abstract objects between a local node, which controls the object, and a remote host that may be only intermittently connected.

A state-synchronization approach is appropriate for tasks like editing a document or using an e-mail or chat application, which control the entire screen and provide their own means of navigation through a document or chat session. But it causes trouble for a task like "cat"-ing a large file to the screen, where the user might rely on having accurate history on the scrollback buffer.

When these semantics are a problem, the user can use a pager such as `less` or `more`, or can use the `screen` or `tmux` utilities, which are essentially pagers for the entire terminal. Future versions of Mosh will allow the user to browse the scrollback history.

The Mosh system runs SSP in each direction, instantiated on two different kinds of objects. From client to server, the objects represent the history of the user's input. From server to client, the objects represent the contents of the terminal window.

2.1 Protocol design goals

SSP's design goals were to:

1. Leverage existing infrastructure for authentication and login, e.g., SSH.
2. Not require any privileged code.
3. At any time, take the action best calculated to fast-forward the remote host to the sender's current state.
4. Accommodate a roaming client whose IP address changes, without the client's having to know that a change has happened.
5. Recover from dropped or reordered packets.
6. Ensure confidentiality and authenticity.

Because SSP doesn't use any privileged code or authenticate users, and key exchange happens out-of-band, its security concerns are simplified. To bootstrap the session, the user runs a script that logs in to the remote host using conventional means (*e.g.*, SSH) and runs the unprivileged server. This program listens on a high UDP port and prints out a random shared encryption key. The system then shuts down the SSH connection and talks directly to the server over UDP.

SSP is organized into two layers. A datagram layer sends UDP packets over the network, and a transport layer is responsible for conveying the current object state to the remote host.

2.2 Datagram Layer

The datagram layer maintains the "roaming" connection. It accepts opaque payloads from the transport layer, prepends an incrementing sequence number, encrypts the packet, and sends the resulting ciphertext in a UDP datagram. It is responsible for estimating the timing characteristics of the link and keeping track of the client's current public IP address.

The security of the system is built on AES-128 in the Offset Codebook (OCB) mode [5], which provides confidentiality and authenticity with a single secret key.

To handle reordered and repeated packets, SSP relies on the principle of idempotency. Each datagram sent to the remote site represents an idempotent operation at the recipient—a "diff" between a numbered source and target state. As a result, unlike Datagram TLS and Kerberos, SSP does not need to maintain a replay cache or other message history state, simplifying the design and implementation.

Client roaming. Every time the server receives an authentic datagram from the client with a sequence number greater than any before, it sets the packet's source IP address and UDP port number as its new "target." As a result, client roaming happens automatically, without the client's timing out or even knowing that it has changed public IP addresses.

Estimating round-trip time and RTT variation. The datagram layer is also responsible for estimating the smoothed round-trip time (SRTT) and RTT variation (RTTVAR) of the connection. Every outgoing datagram contains a millisecond timestamp and an optional "timestamp reply," containing the most recently-received timestamp from the remote host.

We use the algorithm of TCP [7] with three changes:

1. Because every datagram has a unique sequence number, there is no ambiguity between the timestamps of retransmissions of the same payload.
2. SSP adjusts the "timestamp reply" by the amount of time since it received the corresponding timestamp.

Therefore, policies like delayed ACKs do not affect the accuracy of the RTT estimates.

3. We reduce the lower limit on the retransmission timeout to be 50 ms instead of one second. SSH runs over TCP and rarely benefits from fast retransmissions, meaning it generally cannot detect a dropped keystroke in less than a second.

2.3 Transport Layer

The transport layer synchronizes the contents of the local state to the remote host, and is agnostic to the type of objects sent and received.

Transport sender behavior: The transport sender updates the receiver to the current state of the object by sending an Instruction: a self-contained message listing the source and target states and the binary "diff" between them. This "diff" is a *logical* one, calculated by the object implementation. The ultimate semantics of the protocol depend on the type of object, and are not dictated by SSP. For example, for user inputs, the diff contains every intervening keystroke, whereas for screen states, it is only the minimal message that transforms the client's frame to the current one.

Transport sender timing: Because SSP can construct a diff between any two object states, it is not required to send every octet it receives from the host and can modulate the "frame rate" based on network conditions. The minimum interval between frames is set at half the smoothed RTT estimate, so there is about one Instruction in flight to the receiver at any time.[1]

As a result, when a process goes haywire and floods the terminal, network buffers do not fill up and increase latency, so unlike in prior work, Control-C and other interrupt sequences continue to work.

The transport sender uses delayed acks, similar to TCP, to cut down on excess packets. In more than 99.9% of cases in our experiments, a delay of 100 ms was sufficient to let the delayed ACK piggyback on host data.

The server also pauses from the first time its object has changed before sending off an Instruction, because updates to the screen tend to clump together, and it would be wasteful to send off a new frame with a partial update and then have to wait the full "frame rate" interval before sending another. A collection interval of 8 ms was chosen as optimal after analyzing application traces (§4).

SSP sends an occasional heartbeat to allow the server to learn when the client has roamed to a new IP address, and to allow the client to warn the user when it hasn't recently heard from the server. The heartbeat also keeps the connection open when the client is behind a network address translator. We chose an interval of 3 seconds to compromise between responsiveness and the desire to reduce unnecessary chatter.

[1] We cap the maximum frame rate at 50 Hz, roughly the limit of human perception, to save unnecessary traffic on low-latency paths.

3 A Remote Terminal with Speculative Local Echo

To support the Mosh application, we implemented a terminal emulator that obeys the SSP object interface. The client sends all keystrokes to the server, which applies them and maintains the authoritative state of the terminal, which it in turn synchronizes back to the client.

The client intelligently guesses the effect that keystrokes will have on the terminal, and in most cases can speculatively apply such effects immediately. The client observes the success of its predictions to decide how confident to be and whether to display the predictions to the user.

On high-delay connections, we underline unconfirmed predictions so the user doesn't become misled. This underline trails behind the user's cursor and disappears gradually as responses arrive from the server. Occasional mistakes can be removed within an RTT and do not cause lasting effect.

3.1 Implementing the terminal emulator

Mosh's terminal emulator implements the subset of the ISO/IEC 6429/ECMA-48 language [1] used by typical terminal emulators, including the `xterm`, `gnome-terminal`, `Terminal.app`, and `PuTTY` programs for X11, OS X, and Windows. This protocol was popularized by Digital Equipment Corp. in the 1970s and 80s and specifies a series of escape sequences to move the cursor, render characters in bold and colors, erase areas of the screen, etc. The protocol is bidirectional, as the host can query the terminal for its current character position and ask it to identify itself.

3.2 Speculative local echo

Because Mosh operates at the terminal emulation layer and maintains the screen state at both the server and client, it is possible for the client to make predictions about the effect of user keystrokes and later verify its predictions against the authoritative screen state coming from the server.

Most Unix applications operate similarly in response to user keystrokes. They either echo the key at the current cursor location or not. As a result, it is possible to approximate a local user interface for arbitrary remote applications. We use this technique to boost the perceived interactivity of a Mosh session over a high-latency network or one with packet loss.

Our general strategy is for the Mosh client to make an echo prediction each time the user hits a key, but not necessarily to display this prediction immediately.

The predictions are made in groups known as "epochs," with the intention that either all of the predictions in an epoch will be correct, or none will. An epoch begins tentatively, making predictions only in the background. If any prediction from a certain epoch is confirmed by the server, the rest of the predictions in that epoch are immediately displayed to the user, along with any future predictions in the same epoch.

Some user keystrokes are likely to alter the host's echo state from echoing to not, or are otherwise hard to predict, including the up- and down-arrow keys and control characters. These cause Mosh to lose confidence and increment the epoch, so that future predictions are made in the background again.

In practice, this approach accommodates a wide variety of application behaviors, including multi-mode editors like `vi` (which sometimes echo conventionally and sometimes don't), and the possibility that the user might type a command at the prompt (*e.g.*, `passwd`) that stops server-side echoes after the ENTER key is typed.

Because the decision to perform local echo is made entirely based on the application's observed behavior, applications need not be rewritten to accommodate local echo. Unlike prior work, Mosh's local echo works even with full-screen programs (like `emacs`) that put the terminal driver in "raw" mode and do their own echoing.

In typical use, Mosh can display immediately the effects of almost all "typing," which constitutes more than two-thirds of user keystrokes in our captures. The remaining keystrokes are principally "navigation" (such as "n" to move to the next e-mail message in a mail reader), which cannot be predicted locally.

Server-side assistance for prediction evaluation

For the above algorithm to work properly, the Mosh client must be able to reliably determine whether its echo predictions are correct. Early versions of Mosh attempted to do this with the client only, by simply examining whether a predicted echo was present on the screen by the time the Mosh server had acknowledged the corresponding keystroke.

Unfortunately, in trials, we found that applications sometimes take tens of milliseconds after input is presented to them before echoing to the screen. This can lead the Mosh server to acknowledge an input keystroke before the echo is present in the screen state, and causes the client to conclude that its prediction was incorrect, even though the echo is on the way. This produces annoying flicker as the echo is (mistakenly) removed from the screen, then reinstated when it eventually arrives from the server.

Our initial solution to this problem was a client-side timeout, so that a prediction is not considered incorrect until the corresponding keystroke has been acknowledged by the server *and* a certain amount of time has elapsed. Unfortunately, because of network jitter that can delay the eventual echo beyond the timeout, this too produced an annoying number of false-negatives and resulting flicker. (By contrast, setting the timeout long enough to accommodate large amounts of jitter causes mistaken predictions to linger on the screen for too long.)

Our final solution was to implement a server-side timeout of 50 ms, chosen to contain the vast majority of legitimate application echoes on loaded servers, while still fast enough to rapidly detect mistaken predictions. The terminal object that is synchronized to the client contains an "echo ack" field, representing the latest keystroke that has been presented to the application for at least 50 ms and whose effects ought to be reflected in the current screen. The client has no timeouts of its own, and consequently network jitter does not adversely affect the client's ability to evaluate whether a prediction is correct. The cost is increased network traffic, because the server often sends an extra datagram 50 ms after a keystroke to convey the echo ack.

In practice, this has eliminated the flicker caused by false-negatives.

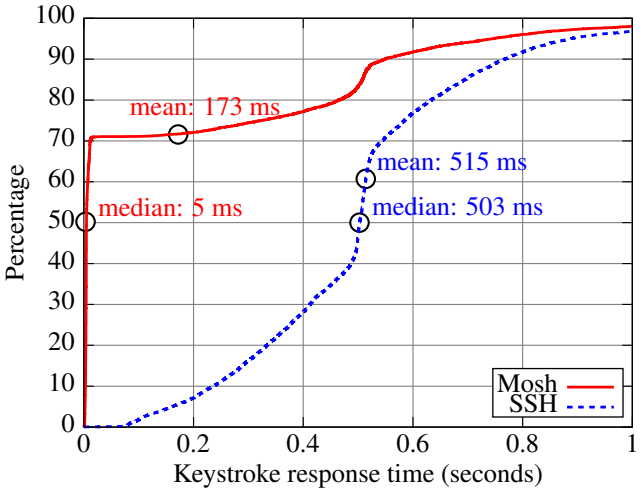
4 Results

We evaluated Mosh using traces contributed by six users, covering about 40 hours of real-world usage and including 9,986 total keystrokes. These traces included the timing and contents of all writes from the user to a remote host and vice versa. The users were asked to contribute "typical, real-world sessions." In practice, the traces include use of popular programs such as the `bash` and `zsh` shells, the `alpine` and `mutt` e-mail clients, the `emacs` and `vim` text editors, the `irssi` and `barnowl` chat clients, the `links` text-mode Web browser, and several programs unique to each user.

To evaluate typical usage of a "mobile" terminal, we replayed the traces over an otherwise unloaded Sprint commercial EV-DO (3G) cellular Internet connection in Cambridge, Mass. A client-side process played the user portion of the traces, and a server-side process waited for the expected user input and then replied (in time) with the prerecorded server output. We sped up long periods with no activity. The average round-trip time on the link was about half a second.

We replayed the traces over two different remote shell applications, SSH and Mosh, and recorded the user interface response latency to each simulated user keystroke, as seen by the user. The Mosh predictive algorithm and

Figure 2: Cumulative distribution of keystroke response times with Sprint EV-DO (3G) Internet service



SSP were frozen prior to collecting the traces and were not adjusted in response to their contents or results.[2]

The cumulative distributions and statistics of keystroke response time are shown in Figure 2. When Mosh was confident enough to display its predictions, the response was nearly instant. This occurred about 70% of the time. But many of the remaining keystrokes were "navigation," such as moving to the next e-mail message, and Mosh cannot make a prediction in these cases. For keystrokes it could not predict, Mosh's latency distribution was similar to that of SSH.

Mosh displayed an erroneous prediction, which it fixed within an RTT, for 0.9% of the keystrokes. These generally occurred because of word-wrap (characters that were printed near the end of a line get moved to the next line at an unpredictable time).

Appropriateness of timing parameters

We also used the user traces to examine our choice of timing parameters for the SSP sender. Here, we assess the choice for the "collection interval": the pause time after receiving a write from the host, in order to collect writes that may be following in close succession. We disregard the possible benefits of speculative local echo and focus on network performance.

Figure 3 shows the artificial delay introduced by the Mosh server on the applications's screen updates in our traces. Recall that the server obeys two rules: always wait at least the frame-rate interval after a previous frame, and always wait at least the "collection interval" after receiving an initial write from the application. This parameter represents a tradeoff: too short could cause the server to send a tiny initial datagram and then wait before sending more data. But too long would hurt the responsiveness of a typical session.

The ideal value depends on how often, empirically, applications tend to wait between their writes. We had initially guessed that a value of 15 ms would be reasonable; based on the results and user feedback, we adjusted that to 8 ms, the minimum of the curve.

Predictive echo on other networks

After tuning the algorithm as discussed above, we evaluated the same user traces replayed over a wireless Internet service loaded with a concurrent TCP download, and a trans-oceanic wired link. Again, Mosh displayed about 70% of the keystrokes instantly, sometimes (but not always) increasing the variance in latencies seen by the user. We summarize these results as follows:

Verizon LTE service in Cambridge, Mass., running one concurrent TCP download:

	Median latency	Mean	σ
SSH	5.36 s	5.03 s	2.14 s
Mosh	$<$ 0.005 s	1.70 s	2.60 s

MIT-Singapore Internet path (to Amazon EC2 data center):

	Median latency	Mean	σ
SSH	273 ms	272 ms	9 ms
Mosh	$<$ 5 ms	86 ms	132 ms

Resilience to high packet loss

We also tested SSP's resilience to packet loss without the benefit of predictive local echo. In general, SSP's delay-based rate control and ability to skip intermediate states allow it to handle links with non-congestive packet loss, which TCP was not designed to handle.

We set up a test network with a Linux-based router, using the `netem` tool to create an artificial RTT of 100 ms and a 29% probability of *i.i.d.* packet loss in each direction, resulting in 50% round-trip packet loss. As expected, TCP[3] produces huge delays because of loss-induced exponential backoffs:

	Median	Mean	σ
SSH	0.416 s	16.8 s	52.2 s
Mosh (no predictions)	0.222 s	0.329 s	1.63 s

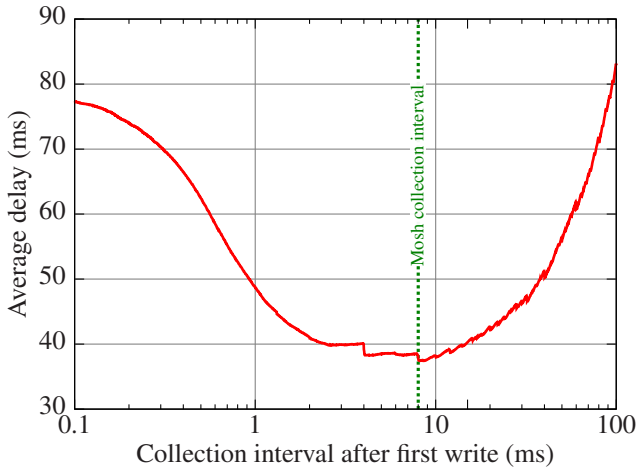
5 Related Work

GNU `screen` and OpenBSD `tmux` are popular "terminal multiplexers" that allow the user to detach from and

[2] We subsequently "unfroze" and modified the Mosh algorithm in response to the data, including moving the collection interval to 8 ms and adding the server-side timeout and "echo ack" feature to reduce false-negative predictions on slow servers (§3.2). These changes improved Mosh in real-world use but would have little effect on this evaluation, because it used a long-delay link with an unloaded server.

[3] Linux 2.6.32 default TCP (`cubic`)

Figure 3: Average protocol-induced delay from varying collection interval (with frame interval of 250 ms)



later reattach to a terminal session. (Graphical remote-desktop programs, such as VNC, also allow reconnection.) `screen` and `tmux` provide several other features, such as multiplexing and scrollback buffers, and are often used concurrently with Mosh.

REX [4] is a remote execution protocol built atop the Self-certifying File System [6]. It uses TCP, but provides automatic roaming in some cases: when the client finds that a TCP connection aborts or a connection timeout occurs, it reinitiates the TCP connection and queues pending data in the mean time. However, it could take several minutes or longer for a TCP connection timeout to occur, especially if the client has no pending data of its own.

Mosh differs from terminal multiplexers and REX in that its roaming is immediate and automatic, using application-level timers that assess the state of connectivity end-to-end. Mosh is also distinct in that it skips over intermediate screen states, even while connected, to accommodate high-latency or loss-prone paths.

Some BSD-style operating systems support the LINEMODE option [2] for TELNET, in which character echoing and line editing is performed by the client. Unfortunately, LINEMODE does not work with programs that put the terminal into "raw" mode, including shells like `bash`, and full-screen applications like `emacs`, `vi`, or `pine`. SSH does not have an equivalent of LINEMODE.

SUPDUP [8] included a Local Editing Protocol in which an entire text editor session could be executed locally and uploaded to the server in batches. SUPDUP required the host application to encode its interactive functionality in the SUPDUP language. Mosh does not require modifications to host applications, but still handles most typing and cursor movement keystrokes.

6 Conclusion

This paper presented the design, implementation, and evaluation of Mosh, a mobile shell that performs well over marginal networks. Mosh handles intermittent connectivity and changes in IP addresses, and provides good interactive performance over long-delay network paths. In our empirical evaluation of 40 hours of keystroke activity from six users, we found that mean and median response times were dramatically reduced on several different types of connections. Mosh achieved this improvement by accurately predicting the response to 70% of user keystrokes. Mosh's wide adoption upon release suggests that it fulfills a previously unmet need among mobile network users.

SSP is a relatively rare example of a gracefully-mobile networking protocol. Today, many programs intended for mobility, including e-mail and chat programs on popular smartphones, cannot cope gracefully with roaming and intermittent connectivity: the very conditions presented by mobile networks. We believe many of these applications would benefit from SSP's design principles.

7 Acknowledgments

We thank Nickolai Zeldovich and Chris Lesniewski-Laas for helpful comments on this work. We also thank Anders Kaseorg, Quentin Smith, Richard Tibbetts, Keegan McAllister, and the users who provided us with keystroke traces. This work was supported in part by NSF grants 1040072 and 0721702.

References

[1] *Control Functions for Coded Character Sets*. ECMA-48 (1991); ISO/IEC 6429:1992.

[2] D. Borman. Telnet linemode option. RFC 1116, 1990.

[3] D. Clark and D. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *SIGCOMM*, 1990.

[4] M. Kaminsky, E. Peterson, D. B. Giffin, K. Fu, D. Mazières, and M. F. Kaashoek. REX: Secure, Extensible Remote Execution. In *USENIX*, June 2004.

[5] T. Krovetz and P. Rogaway. The software performance of authenticated-encryption modes. In *18th Intl. Conf. on Fast Software Encryption*, 2011.

[6] D. Mazières. *Self-certifying File System*. PhD thesis, Massachusetts Institute of Technology, May 2000.

[7] V. Paxson, M. Allman, J. Chu, and M. Sargent. Computing TCP's Retransmission Timer. RFC 6298, 2011.

[8] R. M. Stallman. The SUPDUP Protocol. Technical report, MIT AI Memo 644, 1983.

[9] T. Ylönen. SSH–secure login connections over the Internet. In *6th USENIX Security Symp.*, pages 37–42, 1996.

TROPIC: Transactional Resource Orchestration Platform In the Cloud

Changbin Liu* Yun Mao† Xu Chen† Mary F. Fernández†
Boon Thau Loo* Jacobus E. Van der Merwe†
*University of Pennsylvania †AT&T Labs - Research

Abstract

Realizing Infrastructure-as-a-Service (IaaS) cloud requires a control platform to orchestrate cloud resource provisioning, configuration, and decommissioning across a distributed set of diverse physical resources. This orchestration is challenging due to the rapid growth of data centers, high failure rate of commodity hardware and the increasing sophistication of cloud services. This paper presents the design and implementation of TROPIC, a highly available, transactional resource orchestration platform for building IaaS cloud infrastructures. TROPIC's orchestration procedures that manipulate physical resources are transactional, automatically guaranteeing atomicity, consistency, isolation and durability of cloud operations. Through extensive evaluation of our prototype implementation, we demonstrate that TROPIC can meet production-scale cloud orchestration demands, while maintaining our design goals of safety, robustness, concurrency and high availability.

1 Introduction

The Infrastructure-as-a-Service (IaaS) cloud computing model exemplified by Amazon EC2 [1] provides users on-demand, near-instant access to a large pool of virtual cloud resources such as virtual machines (VMs), virtual block devices, and virtual private networks. The *orchestrations* of the virtual resources over physical hardware, such as provisioning, configuration, and decommissioning, are exposed to the users as a service via programmable APIs. These APIs hide the complexity of the underlying orchestration details.

From the cloud provider's perspective, however, building a robust system to orchestrate cloud resources is challenging in terms of both scale and fault tolerance, as shown by recent studies [8, 13] on the open-source cloud platforms. First, today's large data centers typically run on the scale of over 10,000 machines based on commodity hardware [15]. As such, software glitches and hardware failures including power outages and network partitions are the norm rather than the exception. This unreliability not only impacts the virtual resources assigned to users, but also the controllers that orchestrate the virtual resources. Second, to orchestrate a massively concurrent, multi-tenant IaaS environment, the control logic is inherently complex. In particular, any engineering and service rule must be met while avoiding race conditions. The postmortem from the EC2 outage in April 2011 [10] anecdotally reinforces our arguments: A human error in router configuration that violates an implicit service rule and a race condition in storage provisioning contributed significantly to the prolonged downtime.

To address these challenges, we propose TROPIC, a *transactional* orchestration platform with a unified data model that enables cloud providers to develop complex cloud services with safety, concurrency, robustness and high availability. Specifically, we make the following contributions:

Transactional abstraction. In TROPIC, orchestration procedures are executed as *transactions* with ACID properties (atomicity, consistency, isolation and durability). Transactional semantics provide a clean abstraction to cloud providers to ensure that, orchestrations that encounter unexpected errors have no effect, concurrent orchestrations do not violate safety rules or cause race conditions, and committed orchestrations persist on physical devices. As a result, service developers only need to focus on developing high level cloud services without worrying about the complexities of accessing and managing underlying volatile distributed resources.

Transaction processing. While TROPIC adopts standard database transaction processing techniques such as write-ahead-logging for atomicity and a hierarchical intention locking scheme for concurrency control [20], we propose a two-layer transaction processing stack to cope with the unique challenges in the cloud. In the *logical layer*, each transaction is analyzed for possible resource contention and constraint violations prior to actual execution. This provides early detection of unsafe operations without touching physical resources. Once deemed safe, the transaction is then executed in the *physical layer*. In the presence of resource failures, TROPIC provides reconciliation mechanisms to handle cross-layer inconsistencies.

High availability. TROPIC adopts a highly available decentralized architecture where all components are decoupled to avoid single point of failure. TROPIC runs multiple controllers, and provides efficient recovery mechanisms such that whenever the lead controller fails, another controller can take its place without service disruption while maintaining transactional semantics.

Prototype implementation and evaluation. We have implemented a complete TROPIC prototype deployed on the ShadowNet testbed [14]. We extensively evaluated our prototype using production-scale traces obtained from EC2 and a large US hosting provider, demonstrating that TROPIC is able to manage cloud resources at a

large scale, while ensuring transactional semantics and high availability.

2 TROPIC Overview

2.1 Design Goals

Our objective is to provide a cloud orchestration platform at the scale of at least 100,000 cloud resources (*e.g.,* VMs and block devices) in one data center [5] with the following characteristics.

First, the platform should guarantee that a cloud service is ***safe***, that is, the service's orchestration procedures do not violate any constraints. These constraints reflect service and engineering rules in operation. If violated, an illegal orchestration operation could disrupt cloud services, *e.g.,* spawning a VM on an overloaded compute server, or migrating a VM to an incompatible hypervisor or CPU with different instruction sets. Enforcing these constraints is challenging as it often requires acquiring the states of distributed resources and reasoning about them holistically.

The second goal is that the platform should allow ***high concurrency***, *i.e.,* performing simultaneous execution of massive orchestration procedures safely, especially when they access the same resources. For example, simultaneous spawning of two VMs on the same compute server may exceed the physical memory limit of the server. Concurrency control guarantees that simultaneous execution of orchestration procedures avoids race conditions and permits the platform to scale.

Third, the platform should guarantee that a service is ***robust*** in the presence of unexpected failures. Robustness ensures that failures in an orchestration procedure do not lead to undefined behavior or inconsistent states. This goal is challenging because of high volatility in the cloud environment, caused by software bugs, unstable hardware, transient network disconnections and power outages [9], etc. An orchestration procedure usually involves multiple state changes of distributed resources, any of which can fail due to volatility. For example, spawning a VM has the following steps: clone a VM disk image on a storage server; create a VM configuration on a compute server; set up virtual local-area networks (VLAN), software bridges, and firewalls for inter-VM communication; finally start the VM. During the process, an error at any step would prevent the client from obtaining a working VM. Worse, the leftover configurations in the compute, storage and network components become orphans if not properly cleaned up, which may lead to undefined behavior for future orchestrations.

Our last goal is to guarantee ***high availability*** of cloud services and the platform that manages them. In the era of web-scale applications, unavailability of the cloud platform directly translates to loss of revenue and service degradation for customers. Based on our estimation of Amazon EC2's rate of VM creation (see §6), a mere 10-minute service disruption can result in not fulfilling 1,400 VM spawn operations in a single region. Such disruptions are unacceptable for mission-critical applications.

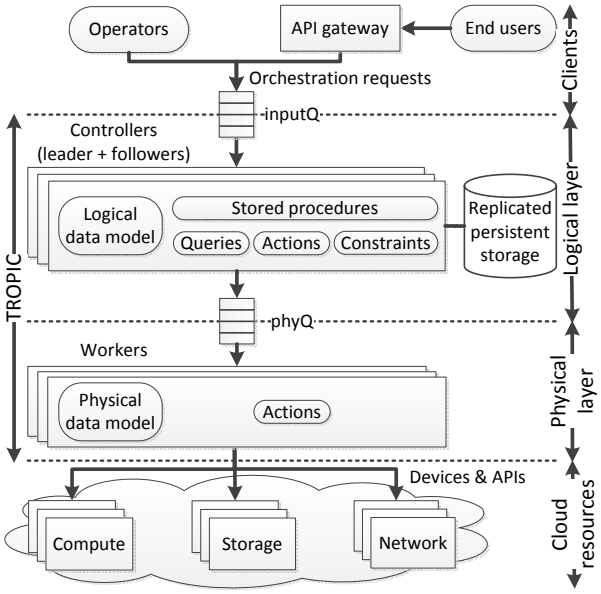


Figure 1: TROPIC architecture

2.2 Architecture

To achieve these design goals, we present the TROPIC platform, which performs *transactional* cloud resource orchestrations. Transactions provide ACID semantics which fit our design goals well: (i) *Safety* is enforced by *integrity constraints* in order to achieve transactional *consistency*; (ii) *Concurrency* is supported by a concurrency control algorithm that permits multiple transactions to execute in parallel while preserving the transactional behavior of *isolation*; (iii) *Robustness* is provided by the *atomicity* and *durability* properties, which guarantee that committed orchestrations persist on physical devices, while orchestrations that encounter unexpected errors have no effect; (iv) *High availability* is enabled by TROPIC's adoption of a decentralized architecture of replicated components.

Figure 1 depicts TROPIC's architecture. The orchestration requests of clients are initiated either directly by cloud operators (*e.g.,* for maintenance), or indirectly by the cloud end users via the API service gateway (*e.g.,* to spawn VMs). Between the clients and cloud resources, TROPIC provides a two-layer orchestration stack with the *controllers* at the *logical layer* and the *workers* at the *physical layer*.

In the logical layer, the controllers provide a unified data model for representing the control states of cloud resources and a domain-specific language for implementing services. The controllers accept orchestration requests and invoke corresponding orchestration

operations—*stored procedures* written in TROPIC's programming language. These stored procedures are executed as transactions with ACID semantics. In the physical layer, the workers straddles the border between the controllers and the physical devices, and provide a physical data model of devices' state. The logical data model contains a *replica* of the physical data model with weak, eventually consistent semantics.

Execution of orchestration operations in the logical layer modifies the logical data model. In the process, actions on physical devices are *simulated* in the logical layer. TROPIC guarantees safety by transitioning the logical model transactionally from one consistent state to another, only after checking that all relevant global safety constraints are satisfied. Resource conflicts are also checked to avoid race conditions. After the checks in the logical layer, corresponding physical actions are executed in the physical layer, invoking device-specific APIs to actually manipulate the devices. Transactional orchestration in both layers is described in detail in §3.

The separation of logical and physical layers is unique in TROPIC and has several benefits. First, updating physical devices' state can take a long time to complete. Simulating changes to physical devices in the logical layer is more efficient than executing the changes directly at the physical layer, especially if there are constraint violations or execution errors. Second, the separation facilitates rapid testing and debugging to explore system behavior and performance prior to deployment (§5). Third, if the logical and physical models diverge (*e.g.,* due to physical resource volatility), useful work can still be completed on consistent parts of the data model, and in the meantime, *repair* and *reload* strategies (§4) are used to reconcile any inconsistencies.

TROPIC adopts a semi-structured hierarchical data model because it handles heterogeneity of cloud resources well. Each tree node is an object representing an instance of an entity. Each entity has associated expressions and procedures for inspecting and modifying the entity: queries, actions, constraints, and stored procedures. A *query* inspects system state in the logical layer and provides *read*-only access to resources. An *action* models an *atomic* state transition of a resource. Each action is defined twice: in the physical layer, the action implements the state transition by calling the device's API, and in the logical layer, the action simulates the state transition on the logical data model. Preferably, an action is associated with a corresponding *undo* action, which is used to roll back a transaction (§3.1). *Constraints* specify service and engineering rules. Constraints support the *safety* property, and TROPIC automatically enforces them at runtime. Orchestration logic is specified as *stored procedures*, composed of queries, actions and other stored procedures to orchestrate cloud resources. [11, 18] provide more details on TROPIC's data model and programming constructs.

2.3 High Availability

The TROPIC architecture is designed with redundancy to avoid single point of failure. First, the components of TROPIC are connected via distributed queue services (`inputQ` and `phyQ`) that are highly available, which reduce the dependency between the components. Second, the persistent storage service is pluggable to any backend system that offers replicated, atomic key-value storage with strong consistency. We adopt ZooKeeper [16] to implement the queues and the storage service (§5).

TROPIC runs multiple controller instances. One of them is the leader, and the rest are followers, decided by a quorum-based leader election algorithm [21]. Only the leader serves transaction executions in the logical layer. When it fails, the followers among themselves elect a new leader, which then resumes execution after restoring the most recent state of the previous leader. TROPIC controllers only maintain state in local memory as a cached copy for performance reasons and can be safely discarded without impacting the correctness of transaction execution. Whenever the lead controller fails at any possible failure points, the new leader elected among the followers are able to restore the state of the controller at failure time, using state from persistent storage. Due to space constraint, we refer interested readers to the technical report [11] for details of the replicated state design and the idempotent recovery protocol.

3 Transactional Orchestration

In this section, we describe TROPIC's transaction execution model, and explain how TROPIC can meet our design goals of safety, concurrency, and robustness, through the enforcement of ACID properties in orchestration operations. Specifically, TROPIC makes the following guarantee: if the logical and physical layers are consistent at the beginning of each transaction, ACID properties can always be enforced in the logical layer. Furthermore, in the absence of cross-layer inconsistency caused by resource volatility, these properties are also enforced in the physical layer. We defer the discussion of inconsistency between the logical and physical layers to §4, and focus on transaction processing here.

We first describe a typical life cycle of a transactional orchestration operation, followed by the execution details in the logical and physical layers. Figure 2 depicts the typical steps in executing a transaction t, from the initial request submitted by a client until t is committed or aborted.

Step 1: initialization. A client issues a transactional orchestration as a call to a stored procedure. The transaction is *initialized* and enqueued to `inputQ`.

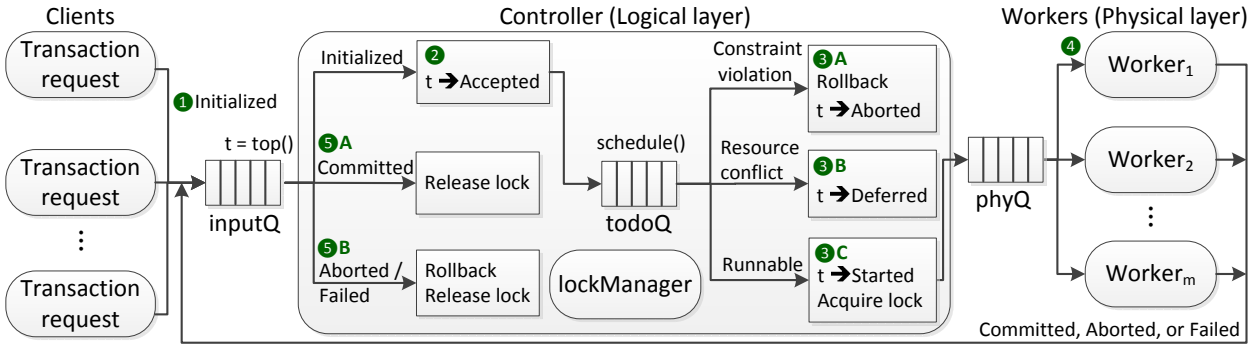


Figure 2: The execution flow of transactional orchestration in TROPIC.

Step 2: acceptance. The controller (leader) accepts *t* by dequeuing it from `inputQ` and enqueues it to `todoQ`.

Step 3: logical execution. The controller is responsible for scheduling accepted transactions, making sure there is no constraint violation or possible race condition, and generating the execution logs for future undo and physical layer execution. All these steps happen in the logical layer and are explained in §3.1.

Step 4: physical execution. Any transaction that has gone through the controller is dequeued from `phyQ` and executed in the physical layer by the physical workers (§3.2). The execution result (*e.g.,* committed or aborted) is enqueued to `inputQ` to notify the controller.

Step 5: cleanup. The controller examines the execution result received from the workers. If it is successful, the transaction state is marked as *committed* and the locks held by the transaction are released (**5A**). Otherwise, if the transaction fails in Step 4, it is marked as *aborted*. The controller then rolls back the logical layer and releases corresponding locks (**5B**).

3.1 Logical Layer Execution

The logical layer execution logic is depicted as Steps **3A**–**3C** in Figure 2. When a transaction *t* is scheduled to execute (`schedule()` in the figure), it is first dequeued from `todoQ`. The controller decides *t* is runnable, if and only if: (i) It does not violate any safety constraints, and (ii) It does not access or modify resources that are being used by outstanding transactions (race conditions). If there is a safety violation, *t* is marked as *aborted* and the controller rolls back the logical layer state (**3A**). If there is a resource conflict, *t* is put back into the front of `todoQ` for subsequent retry (**3B**). Otherwise, *t* is runnable. The controller acquires the locks on related resources, and the transaction state is changed to *started* before *t* is enqueued into `phyQ` (**3C**).

3.1.1 Scheduling

In executing the `schedule()` operation, TROPIC adopts a FIFO queue `todoQ` for fairness and simplicity. It dequeues and schedules a new transaction whenever one of the following conditions is met: (i) A transaction is inserted into an empty `todoQ`; (ii) A transaction is aborted from its logical execution due to a constraint violation; (iii) A transaction finishes its physical execution (either committed or aborted); (iv) A transaction has been identified as runnable and is sent to `phyQ`. More sophisticated scheduling policies are possible (*e.g.,* an aggressive strategy of scheduling transactions queuing behind the one with conflicts). We leave a detailed study of alternative scheduling policies as future work.

3.1.2 Simulation

Once scheduled, instead of directly executing on the physical resources, a simulation step in the logical layer is used to analyze the transaction for possible constraint violations and infer the resources it reads and writes (*i.e.,* queries and actions) for concurrency control. This provides early detection of unsafe operations without touching actual physical resources. Table 1 shows an example transaction for spawning a VM. The transaction consists of 5 actions, which are recorded in an execution log for use in subsequent phases. In simulation, every action within the transaction is applied sequentially, and whenever an action results in a constraint violation, the transaction is aborted. Modifications to the logical layer are rolled back via the *undo* actions in the execution log.

3.1.3 Concurrency Control

TROPIC adopts a pessimistic concurrency-control algorithm based on multi-granularity locking [20]. A lock manager keeps track of the locks acquired by each transaction and detects possible conflicts. New transactions are allowed to run only if their required locks do not conflict with existing locks used by outstanding transactions.

During its execution, a transaction *t* acquires write (read) locks on resource objects used by individual actions (queries). For instance, in table 1, write locks are acquired for each object identified by its resource path. Once these objects and their corresponding lock types are identified, the lock manager acquires read (R) or write (W) locks on the actual object, and *intention locks* (IR/IW)[1] on the ancestors of this object.

[1]Intention locks are commonly used for managing concurrency in hierarchical data structures. They summarize the locking status of descendant nodes, and allow conflicts to be detected higher up the tree.

log record #	resource object path	action	args	undo action	undo args
1	/storageRoot/storageHost	cloneImage	[imageTemplate, vmImage]	removeImage	[vmImage]
2	/storageRoot/storageHost	exportImage	[vmImage]	unexportImage	[vmImage]
3	/vmRoot/vmHost	importImage	[vmImage]	unimportImage	[vmImage]
4	/vmRoot/vmHost	createVM	[vmName, vmImage]	removeVM	[vmName]
5	/vmRoot/vmHost	startVM	[vmName]	stopVM	[vmName]

Table 1: An example of execution log for `spawnVM`

Besides acquiring locks on the resources used by transactions, additional locks are also acquired based on the constraints that impact transactions. When a write operation is performed on an object, we find its highest ancestor that has constraints defined and acquire an R lock on the node. As a result, all its descendants are read-only to other concurrent transactions, hence preventing others from making state changes that may break safety.

3.2 Physical Layer Execution

Once a transaction t is successfully executed in the logical layer, it is ready for actual execution in the physical layer. t is stored in `phyQ` and dequeued by one of the physical workers in Step 4. Executing t in the physical layer involves replaying the execution log generated in the logical layer simulation. If all the physical actions succeed, t is returned as committed. If any action fails, the worker selects the actions that have been successfully executed, identifies the corresponding undo actions, and executes them in reverse chronological order.

To guarantee atomicity of transactions, each action in a transaction must have a corresponding undo action. In our experience, most actions, such as resource allocation and configuration are reversible. Once all undo actions complete, the transaction is returned as aborted. Using the execution log in Table 1 as example, suppose the first four actions succeed, but the fifth one fails. TROPIC reversely executes the undo actions in the log, *i.e.,* record #4, #3, #2 and #1, to roll back the transaction. As a result, the VM configuration and cloned VM image are removed.

If an error occurs during undo in physical execution[2], the transaction is returned as *failed*. The logical layer is still rolled back. However, failures during undo may result in cross-layer inconsistencies between the physical and logical layers.

4 Handling Resource Volatility

In cloud environments, unexpected software and hardware errors (*e.g.,* power glitches, unresponsive servers, misconfigurations, out-of-band access) may occur. We explore mechanisms in TROPIC for dealing with this volatility of resources during transaction execution. TROPIC does not attempt to transparently tolerate failures of the volatile cloud resources. Instead, it makes the best effort to maintain consistency between the logical and the physical layer, by using two reconciliation mechanisms that achieve eventual consistency. In the event of resource failures, TROPIC provides feedback to the cloud operator, in the form of transaction aborts and timeouts, and recovery is handled at higher layers, in accordance with the end-to-end argument [23].

In order for a transaction to execute correctly, the logical layer needs to reflect the latest state of the physical layer. However, achieving cross-layer consistency at all times is improbable given the volatility of cloud resources. To illustrate, consider three scenarios in which inconsistencies occur: (i) During the physical layer execution, an error triggers the rollback procedure, and the execution of an undo action fails. The transaction is terminated as failed, with the logical layer fully rolled back and the physical layer partially rolled back; (ii) An intentional out-of-band change is made to a physical device. For example, an operator may add or decommission a physical resource, or she may log in to a device directly and change its state via the CLI without using TROPIC; (iii) An unintentional crash or system malfunction changes the resource's physical state beyond TROPIC's knowledge. At the scale of large data centers, these events are the norm rather than the exception, and TROPIC must be able to gracefully handle the resulting inconsistencies.

TROPIC adopts an *eventual consistency* model for reconciliation, which allows the two layers to go out of sync in between reconciliation operations. Inconsistency can be automatically identified when a physical action fails in a transaction, or can be detected by periodically comparing the data between the two layers. Once an inconsistency is detected on a node in the data model tree, the node and its descendants are marked *inconsistent* to deny further transactions until the inconsistency is reconciled. Any transactions involving inconsistent data are also aborted with rollback.

The two mechanisms for reconciliation are as follows:

Physical to logical synchronization (reload). States of specified devices are first retrieved from the physical layer and then used to replace the current ones in the logical layer. Similar to normal transaction execution, the controller ensures `reload` is concurrently executed with outstanding transactions while not violating

IW locks conflict with R/W locks, while IR locks conflict with W locks.

[2]We choose to stop executing undo actions in the physical layer once an undo action reports an error, because they might have temporal dependencies.

any constraints. If any constraints are violated, `reload` is aborted.

Logical to physical synchronization (repair). Physical states of devices are also first retrieved. TROPIC then compares the two set of states in the logical and physical layers, and performs corresponding pre-defined actions to repair physical devices. For instance, suppose a compute server is unexpectedly rebooted, resulting in all its running VMs being powered off. By comparing the VM states in two layers — one "running" and the other "stopped", `repair` will execute multiple `startVM` actions to start the powered-off VMs. After `repair` the logical layer is intact and hence no constraint violation should be found in this process.

In the event that `reload` and `repair` operations do not succeed due to hardware failures, the failed resources are marked as *unusable*, and future transactions are prevented from using them.

Given that `repair` and `reload` operations are expensive, we do not run them at the beginning of each transaction. Instead, `repair` is periodically invoked at a frequency customized by cloud operators, and `reload` is called when devices are added to or decommissioned from TROPIC.

Another source of error induced by resource volatility is the indefinite stalling of a transaction. This prevents the transaction from completing (either to a committed, aborted, or failed state) within a bounded period of time. To handle unresponsive transactions, TROPIC provides two mechanisms, by sending either *TERM* or *KILL* signals[3]. A *TERM* signal aborts the outstanding transaction via rollback with graceful cleanups at both the logical and physical layer (*e.g.,* undo actions, lock releasing) so that cross-layer consistency is maintained. A *KILL* signal makes the controller always immediately abort the transaction, but only in the logical layer. Any resulting cross-layer inconsistencies are then reconciled using `repair`.

5 Implementation

We have implemented a prototype of TROPIC. We briefly describe some of our implementation choices and outline a cloud service developed based on top of TROPIC.

We have chosen Python as our implementation language and the prototype of TROPIC is implemented in 11K lines of code. We use ZooKeeper [16] as the distributed coordinator to implement leader election and distributed queues (`inputQ` and `phyQ`). ZooKeeper provides highly available coordination services to large-scale distributed systems. We also unconventionally use ZooKeeper as a highly available persistent storage engine for storing the transaction states and logs.

TROPIC offers a *logical-only mode* to simplify testing and debugging. In this mode, we bypass the physical resource API calls in the workers, and instead focus on various scenarios in the logical layer execution. In this mode, we can easily plug in arbitrary configurable resource types and quantities to study their possible impact on TROPIC. Our experiments in §6 heavily use the logical-only mode to explore TROPIC performance under large scale of diverse cloud resources.

Using TROPIC we have developed a cloud service named *TCloud*. TCloud is deployed in a single data center and has features similar to Amazon EC2. It allows end users to spawn new VMs from disk images, and start, stop, and destroy these VMs. The operator can migrate VMs between hosts to balance or consolidate workloads. The data center provides storage servers that export block devices via the network, compute servers that allocate VMs, and a programmable switch layer with VLAN features. Specifically, we use GNBD [4] and DRBD [22] over the Linux logical volume manager (LVM) as storage resources, Xen [12] as compute resources, and Juniper routers as network resources.

6 Evaluation

In this section, we present the evaluation of our TROPIC prototype implementation. We emulate cloud orchestration workloads using traces from two production systems. The first trace (*EC2*) is inferred from Amazon EC2 and is representative of the rate at which VMs are created within a large scale cloud environment. We use this trace to evaluate the *performance* of TROPIC, in particular its ability to achieve the design goal of *high concurrency*, as defined in terms of metrics such as transaction overhead, latency and throughput.

The EC2 trace is limited to VM spawn operations, which does not capture all the complexities involved in cloud orchestration. We therefore make use of a second workload (*hosting*) derived from the traces obtained from a large US hosting provider. We use this second workload to evaluate the *safety*, *robustness* and *high availability* aspects of TROPIC.

Throughout the experiments, we run three TROPIC controllers, instantiated on three physical machines. Each machine has 32GB memory with 8-core $3.0GHz$ Intel Xeon E5450 CPU processors and runs CentOS Linux 5.5, interconnected via Gigabit Ethernet. TROPIC runs one physical worker with multiple threads[4] which co-locates with one of the physical machines. As the distributed coordinator and replicated persistent storage, three ZooKeeper instances reside on the same set of physical machines.

6.1 Performance

Workload. The EC2 workload used to evaluate the performance of TROPIC was collected in July 2011. We

[3] Analogous to `SIGTERM` and `SIGKILL` signals to a POSIX-compliant process.

[4] TROPIC can of course run multiple workers, but doing so does not alter the conclusions drawn from our evaluation results.

measured the number of newly launched VM instances over a week period in the US-east region using the methodology described by RightScale [2]. Specifically, we created a VM instance every 60 seconds and recorded the VM ID. The ID (after decoding) is unique and the distance between any two consecutive IDs reflects the quantity of VMs spawned in between. Figure 3 shows the measured workload in a 1-hour period. The workload in total contains 8417 VM spawnings, with an average of 2.34 per second and a peak of 14.0 at 0.8 hours. We choose this time window because it has a typical average VM launch rate (2 VMs/s) and also the highest peak rate during the week we observed.

Controller CPU overhead. Next we use the 1-hour EC2 trace to inject the synthetic workload in TROPIC, by submitting VM spawn transactions every second. To simulate a large-scale cloud environment, we run TROPIC in the *logical-only* mode (§5) with 12,500 compute servers. Each server has 8 VMs, totaling 100,000 VMs (our target scale). 3,125 storage servers are used to hold the VM images, *i.e.,* 4 compute servers share a storage server. To explore the behavior of TROPIC under higher load, we further multiply the EC2 workload from 2 times (2×) to 5 times (5×), and measure the CPU utilization of the controller (leader) as shown in Figure 4.

We observe that the CPU utilization is synchronized with the workloads. As the workloads scale up, CPU utilization rises linearly. However, even during the peak load of 5× EC2 workload, the CPU only reaches as high as 54.0%. Additionally, we measure the memory footprint of TROPIC controller. It is relatively stable, at around 5.4% (of 32GB) for all workloads. We note that the dominant factor contributing to the memory footprint is the quantity of all managed cloud resources, instead of the active workload. After 0.8 hours the CPU peaks of 4× and 5× EC2 workloads retain longer than the workload peak. It is because during the period TROPIC reached the limit of transaction throughput, and hence experienced delays in processing each transaction.

Transaction latency. Figure 5 shows a detailed breakdown of per-transaction latency results, in the form of a cumulative distribution function (CDF). We define the transaction latency as the time duration from the submission of a transaction until it is successfully committed or aborted. In Figure 5 the median latency is less than 1s for all the workloads. For 1× workload, the latency is almost negligible. As expected, 4× and 5× workloads have higher transaction latency, mostly as a result of the workload spike from 0.8 to 1.0 hours.

We further investigate the factors affecting performance bottlenecks of TROPIC under high load. Our experimental results [11] indicate that the dominant overhead comes from ZooKeeper API calls (*I/O*) instead of TROPIC logical layer simulation (*CPU*). To analyze scalability, we measure transaction throughput as the quantity of resources and transactions (load input) scales up. Our results demonstrate that TROPIC transaction throughput stays constant as the number of resources and transactions increases. This is due in part to our efficient implementation and optimizations. Moreover most of the factors affecting throughput (*e.g.,* locking overhead, ZooKeeper queue management) incur constant costs. The main bottleneck of TROPIC lies instead with physical memory used to store the data model. Given our specific hardware, the maximum resource scale TROPIC can handle is 2 million VMs.

6.2 Safety

To evaluate the design goals of safety, robustness and high availability of TROPIC, we use the *hosting* workload derived from a data center trace obtained from a large US hosting provider. Unlike the EC2 workload, it involves a more complex set of orchestration procedures. From the trace, we generate the hosting workload consisting of VM *Spawn*, *Start*, *Stop* and *Migrate* operations to mimic a realistic TCloud deployment (§5).

We first use the hosting workload to evaluate the overhead of enforcing safety constraints in TROPIC. We consider two representative constraints featured in TCloud: (1) VM type constraint: VM migration cannot be performed across hosts running different hypervisors; and (2) VM memory constraint: aggregated VMs memory cannot exceed the host's capacity. We focus primarily on per-transaction CPU overhead, since the bulk of constraint checking overhead happens at the logical layer. Our experimental results [11] show the logical layer overhead incurred in checking the above constraints is reasonably low (less than 10ms).

6.3 Robustness

In order to evaluate TROPIC's performance in guaranteeing robustness via transaction atomicity, we carry out two error scenarios, drawn from our experiences in deploying TCloud: VM spawning error and VM migration error. In our experiment, we measure the logical layer overhead of TROPIC in performing transaction rollback in the presence of the previous two errors. To emulate the errors, we execute TROPIC with the hosting workload, and randomly raise exceptions in the last step of VM spawn and migrate. In all our experiments [11], on a per-transaction basis, the logical layer operations complete in less than 9ms. This demonstrates that TROPIC is efficient at handling transaction errors and rollback.

6.4 High Availability

Finally, we evaluate the ability of TROPIC to recover in the presence of controller failures. We deploy TCloud running the hosting workload on the ShadowNet testbed using machines geographically dispersed across the US. Our results [11] demonstrate that TROPIC can recover

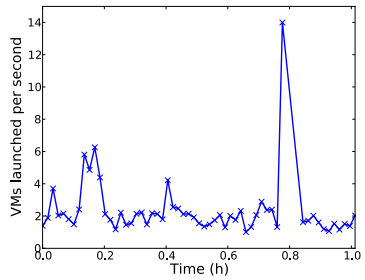


Figure 3: VMs launched per second (EC2 workload).

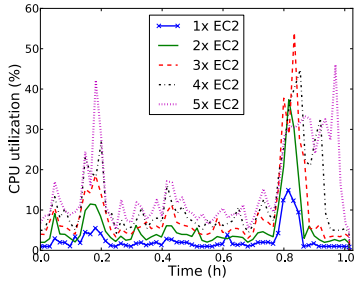


Figure 4: Controller CPU utilization (EC2 workload).

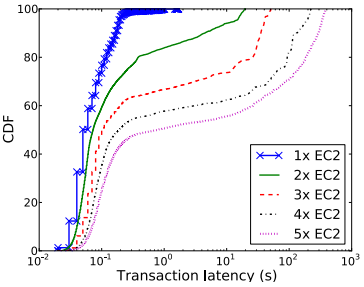


Figure 5: CDF of transaction latency (EC2 workload).

quickly (within 12.5 seconds) to resume processing ongoing transactions in the presence of controller failures. No transaction submitted during the recovery time is lost. The recovery time is dominated by ZooKeeper's failure detection time as the heart-beat interval, suggesting that the recovery time can be reduced by adopting a more aggressive failure detection in ZooKeeper.

7 Related Work

Besides proprietary cloud orchestration platforms from commercial IaaS providers such as Amazon EC2 [1], open-source cloud control platforms, such as OpenStack [6] and Eucalyptus [3], have predefined cloud service models embedded in their implementations. However, none of them provide transactional resource management at the granularity of cloud operations. In contrast, TROPIC is not simply a cloud service, but a general-purpose programming platform to build safe, robust, and highly available cloud services.

Transaction processing has been studied in database area for decades [20]. As a programming paradigm, it has also received more attentions recently from the systems community. These include transactional OS system call APIs [19], file systems [25], and user-level library [24] for lightweight data management. Puppet [7] is a data center automation and configuration management framework. Puppet has a transactional layer, but not in the sense of enforcing ACID properties. Autopilot [17] is a data center software management infrastructure for automating software provisioning, monitoring and deployment. It has repair actions similar to TROPIC, but it does not provide a transactional programming interface. TROPIC borrows ideas from these prior work, such as undo log based rollback, multi-granularity locking. However, the transactional orchestration in TROPIC is unique, in dealing with the logical and physical layer separation and volatile nature of cloud resources, with a "safety-first" mindset.

8 Conclusion

This paper presents TROPIC, a transactional framework for service providers to safely and efficiently orchestrate cloud resources. Our experience in building cloud services on top of TROPIC demonstrates its usability in handling errors, enforcing constraints, and eliminating race conditions. The evaluation of the TROPIC prototype shows its capability to support workload with high degrees of concurrency, provide high availability with low overhead, and ensure the transactional semantics of cloud operations.

9 Acknowledgments

This work is supported in part by NSF grants CCF-0820208, NSF CNS-0845552, and NSF CNS-1040672.

References

[1] Amazon Elastic Compute Cloud (EC2). `http://aws.amazon.com/ec2/`.
[2] Amazon Usage Estimates. `http://bit.ly/poIkqk`.
[3] Eucalyptus Cloud Computing Infrastructure. `http://eucalyptus.com/`.
[4] GNBD Project. `http://sourceware.org/cluster/gnbd/`.
[5] How Big is Amazon's EC2? `http://bit.ly/rjy4Zp`.
[6] OpenStack. `http://openstack.org/`.
[7] Puppet: A Data Center Automation Solution. `http://puppetlabs.com`.
[8] Running 200 VM instances on OpenStack Compute. `http://bit.ly/n7LyMx`.
[9] Summary of the Amazon EC2, Amazon EBS, and Amazon RDS Service Event in the EU West Region. `http://bit.ly/r7aXXR`.
[10] Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. `http://bit.ly/jFdKAR`.
[11] TROPIC: Transactional Resource Orchestration Platform In the Cloud. Extended Technical Report. AT&T TechDoc TD-100446. `http://www.netdb.cis.upenn.edu/papers/tropic_tr.pdf`.
[12] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, 2003.
[13] R. Bradshaw and P. T. Zbiegiel. Experiences with eucalyptus: deploying an open source cloud. In *LISA*, 2010.
[14] X. Chen, Z. M. Mao, and J. Van der Merwe. ShadowNet: A Platform for Rapid and Safe Network Evolution. In *Proc. USENIX ATC*, 2009.
[15] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The cost of a cloud: research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39:68–73, 2008.
[16] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *USENIX ATC*, 2010.
[17] M. Isard. Autopilot: automatic data center management. *SIGOPS Oper. Syst. Rev.*, 41(2):60–67, 2007.
[18] C. Liu, Y. Mao, J. Van der Merwe, and M. Fernandez. Cloud Resource Orchestration: A Data-Centric Approach. In *CIDR*, pages 1–8, 2011.
[19] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating system transactions. In *SOSP*, 2009.
[20] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, third edition, 2002.
[21] B. Reed and F. P. Junqueira. A simple totally ordered broadcast protocol. In *LADIS*, pages 2:1–2:6, 2008.
[22] P. Reisner. DRBD - Distributed Replication Block Device. In *9th International Linux System Technology Conference*, 2002.
[23] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, November 1984.
[24] R. Sears and E. Brewer. Stasis: Flexible transactional storage. In *OSDI*, 2006.
[25] R. P. Spillane, S. Gaikwad, M. Chinni, E. Zadok, and C. P. Wright. Enabling transactional file access via lightweight kernel extensions. In *FAST*, 2009.

Trickle: Rate Limiting YouTube Video Streaming

Monia Ghobadi*
University of Toronto
monia@cs.toronto.edu

Yuchung Cheng Ankur Jain Matt Mathis
Google
{*ycheng, jankur, mattmathis*}*@google.com*

Abstract

YouTube traffic is bursty. These bursts trigger packet losses and stress router queues, causing TCP's congestion-control algorithm to kick in. In this paper, we introduce Trickle, a server-side mechanism that uses TCP to *rate limit* YouTube video streaming. Trickle paces the video stream by placing an upper bound on TCP's congestion window as a function of the streaming rate and the round-trip time. We evaluated Trickle on YouTube production data centers in Europe and India and analyzed its impact on losses, bandwidth, RTT, and video buffer under-run events. The results show that Trickle reduces the average TCP loss rate by up to 43% and the average RTT by up to 28% while maintaining the streaming rate requested by the application.

1 Introduction

YouTube is one of the most popular online video services. In fall 2011, YouTube was reported to account for 10% of Internet traffic in North America [1]. This vast traffic is delivered over TCP using HTTP progressive download. The video is delivered just-in-time to the video player, so when the user cancels a video, only a limited quantity of data is discarded, conserving network and server resources. Since TCP is designed to deliver data as quickly as possible, the YouTube server, *ustreamer*, limits the data rate by pacing the data into the connection. It does so by writing 64kB data blocks into the TCP socket at fixed intervals. Unfortunately, this technique, termed **application pacing**, causes bursts of back-to-back data packets in the network that have several undesirable side effects. These bursts are responsible for over 40% of the observed packet losses in YouTube videos on at least one residential DSL provider [2]. This problem is not specific to YouTube videos. Similar rate limiting techniques are implemented in other popular video websites [6], and all are expected to experience similar side effects. For example, Netflix sends bursts as large as 1 to 2MB.

As an alternative to application pacing, we present Trickle to rate limit TCP on the server side. The key idea in Trickle is to place a dynamic upper bound on the congestion window (*cwnd*) such that TCP itself limits both the overall data rate and maximum packet burst size using ACK clocking. The server application periodically computes the *cwnd* bound from the network Round-Trip Time (RTT) and the target streaming rate, and uses a socket option to apply it to the TCP socket. Once it is set, the server application can write into the socket without a pacing timer and TCP will take care of the rest. Trickle requires minimal changes to both server applications and the TCP stack. In fact, Linux already supports setting the maximum congestion window in TCP.

The main contribution of this paper is a simple and generic technique to reduce queueing and packet loss by smoothly rate-limiting TCP transfers. It requires only a server-side change for easy deployment. It is not a special mechanism tailored only for YouTube. As TCP has emerged to be the default vehicle for most Internet applications, many of them require certain kinds of throttling. The common practice, application pacing, may cause burst losses and queue spikes. Through weeks-long experiments on production YouTube data centers, we found that Trickle reduces the packet losses by up to 43% and RTTs by up to 28% compared to the application pacing.

2 Design and Implementation

The YouTube serving infrastructure is complicated, with many interacting components, including load balancing, hierarchical storage, multiple client types and many format conversions. All YouTube content delivery uses the same server application, called **ustreamer**, independent of client type, video format or geographic location. Us-

*Ghobadi performed this work on an internship at Google mentored by Cheng.

treamer supports progressive HTTP streaming and range requests. Most of the time, a video is delivered over a single TCP connection. However, certain events, such as skipping forward or resizing the screen can cause the client to close one connection and open a new one.

The just-in-time video delivery algorithm in YouTube uses two phases: a *startup* phase and a *throttling* phase. The startup phase builds up the playback buffer in the client, to minimize the likelihood of player pauses due to the rebuffering (buffer under-run) events. Ustreamer sends the first 30 to 40 seconds of video as fast as possible into the TCP socket, like a typical bulk TCP transfer. In the throttling phase, ustreamer uses a token bucket algorithm to compute a schedule for delivering the video. Tokens are added to the bucket at 125% of the video encoding rate. Tokens are removed as the video is delivered. The delay timer for each data block (nominally 64kB) is computed to expire as soon as the bucket has sufficient tokens. If the video delivery is running behind for some reason, the calculated delay will be zero and the data will be written to the socket as fast as TCP can deliver it. The extra 25% added to the data rate reduces the number of rebuffering events when there are unexpected fluctuations in network capacity, without incurring too much additional discarded video.

The just-in-time delivery described above smoothes the data across the duration of each video, but it has an unfortunate interaction with TCP that causes it to send each 64kB socket write as 45 back-to-back packets. The problem is that bursts of data separated by idle periods disrupt TCP's self clocking. For most applications TCP data transmissions are triggered by the ACKs returning from the receiver, which provide the timing for the entire system. With YouTube, TCP typically has no data to send when the ACKs arrive, and then when ustreamer writes the data to the socket it is sent immediately, because TCP has unused *cwnd*.[1] These bursts can cause significant losses, e.g., 40% of the measured YouTube losses in a residential ISP [2]. Similar issues have also been reported by YouTube network operations and other third parties. Worse yet, these bursts also disrupt latency-sensitive applications by incurring periodic queue spikes [11, 17]. The queueing time of a 64kB burst over an 1Mbps link is 512ms. Our goal is to implement just-in-time video delivery using a mechanism that does not introduce large bursts and preserves TCP's self clocking.

A quick solution to the burst problem is to use smaller blocks, e.g., 16kB instead of 64kB. However, this would quadruple the overhead associated with write system calls and timers on the IO-intensive YouTube servers. A better solution is to implement a rate limit in TCP itself. One approach could leverage TCP flow control by fixing the receiver's window (*rwin*) equal to the target streaming rate multiplied by RTT. Once the receiver fixes *rwin*, the ustreamer can write the video data into the socket as fast as possible. The TCP throughput will be limited by the receive window to achieve the target streaming rate. However, this receiver-based approach is not practical because YouTube does control user browsers. Our solution, in contrast, sets an upper-bound on *cwnd* of $target\_rate \times RTT$, where the *target_rate* is the target streaming rate of a video in the throttling phase. Fortunately, Linux already provides this feature as a per-route option called *cwnd_clamp*. We wrote a small kernel patch to make it available as a per-socket option.

The above idea encounters two practical challenges: (1) **Network congestion causing rebuffering**. Following a congestion episode, ustreamer should deliver data faster than the target rate to restore the playback buffer. Otherwise, the accumulated effects of multiple congestion episodes will eventually cause rebuffering events where the codec runs out of data. The current application pacing avoids rebuffering after congestion events implicitly: when TCP slows down enough to stall writes to the TCP socket, ustreamer continues to accumulate tokens. Once the network recovers, ustreamer writes data continuously until the tokens are drained, at which point the average rate for the entire throttled phase matches the target streaming rate. On the other hand, clamping the *cwnd* will not allow such catch-up behavior. (2) **Small** *cwnd* **causing inefficient transfers**. For instance, sending at 500kbps on a 20ms RTT connection requires an average window size of 1250 bytes, which is smaller than the typical segment size. With such a tiny window all losses must be recovered by timeouts, since TCP fast recovery requires a window of at least four packets [5]. Furthermore, using a tiny window increases the ustreamer overhead because it defeats TCP segmentation offload (TSO) and raises the interrupt processing load on the servers.

Trickle starts from the basic design of cwnd limitation and addresses both challenges. Algorithm 1 presents it in pseudocode.[2] After the startup phase, the ustreamer determines the streaming rate, R, based on the video encoding rate. When the data from the cache system arrives, the ustreamer gets the RTT and the Maximum Segment Size (MSS) of the connection (using a socket option) to compute the upper bound of the `clamp`. But before applying the `clamp` on the connection, ustreamer takes two precautions to address the challenges described previously. First, to deal with transient network congestion, ustreamer adds some headroom to the `clamp`. In the experiments we used 20% headroom but we also get simi-

[1] In some cases using congestion window validation [13] would force TCP to do new slow starts after idling over several retransmission timeouts (RTO). This would not always be useful in YouTube as the application writes are more frequent.

[2] An animated demo of Trickle is available at `http://www.cs.toronto.edu/~monia/tcptrickle.html`

Algorithm 1: Trickle algorithm in throttling phase

```
R = target_rate(video_id)
while (new data available from the cache)
  rtt = getsockopt(TCP_INFO)
  clamp = rtt * R / MSS
  clamp = 1.2 * clamp
  goodput = delivered / elapsed
  if goodput < R:
    clamp = inf
  if clamp < 10:
    clamp = 10
    write_throttle = true
  setsockopt(MAX_CWND, clamp)
  if write_throttle:
    throttles writes at rate R
  else:
    write all available data
```

lar results with 5%. If the link is experiencing persistent congestion and/or does not have enough available bandwidth, the ustreamer removes the `clamp` by setting it to infinity and let TCP stream as fast as possible. When the goodput has reached R, the ustreamer will start clamping again. Second, the ustreamer never reduces `clamp` below 10 MSS to address the second constraint. Studies have shown that Internet paths can tolerate burst of this size [7, 9]. However, doing this also increases the streaming rate beyond R to 10×MSS/RTT. Ustreamer thus throttles writes to rate R using application pacing. Unlike the original ustreamer, however, our modified ustreamer never causes bursts of more than 10 packets. Finally, the ustreamer clamps the *cwnd* via a socket option. If the write throttling is enabled, it throttles the write at rate R. Otherwise it writes all data into the socket.

3 Experiments

We performed live experiments to evaluate Trickle on production YouTube data centers. We begin this section with the methodology to compare Trickle and existing application pacing, followed by details of the data centers. Then, we present the measurement results that validate the A/B test setup and Trickle implementation. The first goal is to evaluate if Trickle reduces burst drops and queueing delays. The second goal is to ensure the streaming quality is as good or better than current systems. This is done by measuring the average streaming rate and the rebuffering events. In addition to comparing with current systems, we also compare with the simplest solution, namely reducing the block size from 64kB to 16kB. We ran 4-way experiments by splitting the servers into four groups: (1) Baseline1: application pacing with 64kB blocks, (2) Baseline2: application pacing with 64kB blocks, (3) Trickle, and (4) shrunk-block: application pacing with 16kB blocks. In order to make an apples-to-apples comparison, new TCP connections (video requests) are randomly assigned to the servers in different experiment groups. Thus each experiment group received similar distributions of users and video requests. We use two baseline groups to estimate the confidence level of the particular metric evaluated in our analyses. All servers use the standard Linux 2.6 kernel with CUBIC [12] congestion control. TCP configuration details can be found in Dukkipati *et al.* [8]. For every connection, we recorded statistics including video ID, IP and ports, bytes sent and retransmitted, RTTs, and rebuffering events in both phases. We further filtered the connections that never enter throttling phase (short video playbacks less than 30 seconds). We ran experiments for 15 days during the fall of 2011 in two data centers representing relative extremes of user network conditions: DC1 in Western Europe and DC2 in India. We compare the statistics in the control variables in the experiments to validate the A/B test setup. We verified that each experiment group has roughly the same number of flows within each data center. Moreover, we ensured that flow sizes, flow completion times, and video streaming rate in each group are also similar across different groups in the same data center. Due to lack of space, we refer the interested reader to our technical report for more details [10].

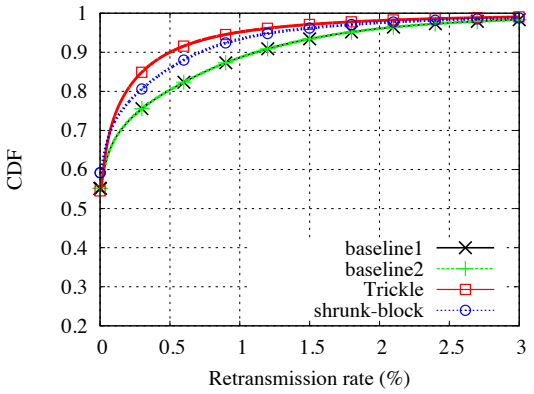


Figure 1: CDF of retransmission rate in the throttling phase.

3.1 Packet Losses

The most important metric is packet loss because Trickle is designed to reduce burst drops. Since losses can not be accurately estimated in live server experiments [4], we use retransmissions to approximate losses. Figure 1 plots the CDF of flow retransmission rate in the throttling phase for DC1. As shown, the Trickle curve is consistently above all three lines, indicating that it successfully

	DC1		DC2	
BW (Mbps)	% flows	avg. retrans. imprv.	% flows	avg. retrans. imprv.
< 0.5	1%	5%	39%	3%
$0.5 - 1$	3%	23%	26%	8%
$1 - 2$	10%	40%	17%	32%
$2 - 4$	28%	53%	9%	47%
$4 - 8$	35%	56%	6%	47%
≥ 8	23%	53%	3%	44%

Table 1: The retransmission rate improvement bucketed by user bandwidth.

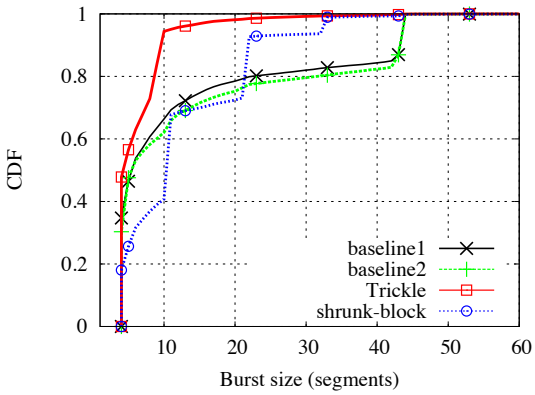


Figure 2: CDF of burst size in DC1.

lowers the retransmission rate consistently compared to the other three groups. In Trickle, 90% of connections experience retransmission rate lower than 0.5%, while 85% have this behavior using shrunk-block and 80% in baselines. On average, Trickle reduces the average retransmission rate by 43% and 28% compared to the baselines and shrunk-block experiments groups, respectively. Overall, Trickle effectively reduces the drop rate compared to application pacing using 64kB or 16kB block sizes. Unlike the results in DC1, however, we measured that all four groups in DC2 have similar retransmission rate distributions. This is because most DC2 users have insufficient bandwidth to stream at the target rate. As described in Section 2, Trickle will detect the delivery is falling behind the target rate and stop clamping the *cwnd*. Therefore connections are not rate-limited by the ustreamer throttling but by the network bandwidth and behave like bulk download in all experiment groups.

To demonstrate the effect of bandwidth, we show the average reduction of the retransmission rate between Trickle and baseline1 bucketed by flows's BW in Table 1. Given that the average target rates are 677kbps and 604kbps in DC1 and DC2 respectively, the table shows that users with low bandwidth do not benefit from Trickle. On the other hand, about half of packet losses can be avoided in for high-bandwidth users in YouTube using Trickle.

3.2 Burst Size

The previous results show that Trickle effectively reduces the loss rate. In this section, we demonstrate that the reduction is achieved by Trickle sending much smaller bursts. We randomly sampled 1% of flows and collected tcpdump packet traces at the server to investigate the burstiness behavior. Following the convention of prior studies [7, 14], we use packets instead of bytes to measure the burst size. We use the same definition of micro-burst as Blanton *et al.* [7]; a burst is a sequence of four or more data packets with inter-arrival time less or equal to 1 millisecond. We use four or more packets because TCP congestion control, e.g., slow start, naturally sends bursts up to three packets. Figure 2 plots the burst sizes in DC1. Intuitively, most bursts in the two baseline groups should be about 43 packets (64kB) but in reality only 20% are. This mismatch is due to packet losses and TCP congestion control. After a packet is lost, TCP congestion control reduces *cwnd* and gradually increases it while streaming new data. These *cwnd* changes fragment the intermittent 64kB application writes. The shrunk-block curve exhibits interesting steps at 12, 23, and 34 packets corresponding to 16, 32, and 48 kB block sizes, respectively. These steps suggest that either the application and/or the kernel (TCP) is bunching up the writes. We then discovered that the ustreamer token bucket implementation does not pause the write for intervals less than 100ms to save timers. For a large portion of the flows, ustreamer continues to write 16kB blocks due to this special handling. Lastly, in Trickle 94% of bursts are within 10 packets, because DC1 users have short RTT such that most videos require less than a 10 packet window to serve. As described in Section 2, Trickle lower-bounds the clamp to 10 packets to avoid slow loss recovery. The remaining 6% of bursts over 10 packets are contributed by either high RTT, or high resolution videos, or other factors that cause Trickle to not clamp the *cwnd*. In summary, over 80% of bursts in Trickle are smaller than the other mechanisms.

3.3 Queueing Delay

Sending smaller bursts not only improves loss rate, it may also help reduce the maximum queue occupancy on bottleneck links. It is certainly not uncommon for users to watch online videos while surfing the Web at the same time. Since networks today are commonly over-buffered [22], shorter queue length improves the latency

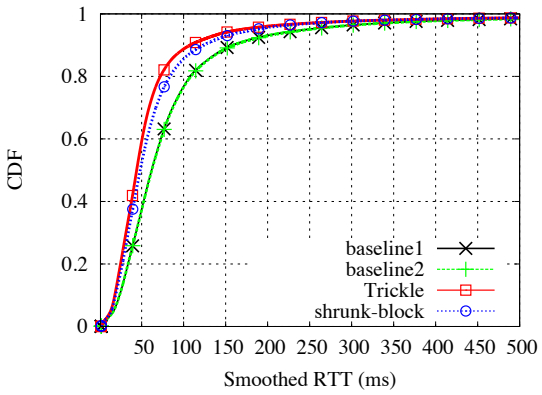


Figure 3: CDF of the smoothed RTT (*srtt*) samples in DC1.

	DC1		DC2	
	rebuff. freq. (1/s)	rebuff. chance (%)	rebuff. freq. (1/s)	rebuff. chance (%)
baseline1	0.0005	2.5%	0.005	26%
baseline2	0.0005	2.5%	0.005	26%
Trickle	0.0005	2.5%	0.005	26%
shrunk-block	0.0005	2.5%	0.005	27%

Table 2: A comparison of rebuffering frequency and rebuffering chance.

of interactive applications sharing the link. We evaluate the impact of queue length by studying the RTT measurements in TCP, due to the lack of direct information of the bottleneck queues. Recall that a RTT sample in TCP includes both the propagation delay and queueing delay. Given that the four experiment groups receive similar load, the propagation delay distribution in each group should be close. Each video stream often has hundreds to thousands of RTT samples partly because Linux samples RTT per ACK packet. In order to reduce the sample size, we instead use the smoothed RTT (*srtt*) variable at the end of the connection. Since *srtt* is a weighted moving average of all the RTT samples, it should reflect the overall queueing delay during the connection.

Figure 3 plots the CDF of the *srtt* samples for DC1. On average, the *srtt* of connections in the Trickle group is 28% and 10% smaller than the connections in the baselines and shrunk-block groups, respectively. In DC2, the improvement over baseline is only 7% and 1%. The reason is similar to the analysis in Section 3.1: throttling is seldom activated on the slow links in DC2. We measured that the links in India are alarmingly over-buffered: 20% of the *srtt* samples were over 1 second while 2% were over 4 seconds. While these videos are likely being streamed in the background, the interactive applications sharing the same bottleneck queue certainly will suffer extremely high latency. In summary, for fast networks, Trickle connections experience much lower queueing delays, which should improve interactive application latencies. For slow users, the solution is to use Trickle but serve at a lower rate (lower resolution video).

3.4 Rebuffering

Rebuffering happens when a reduction in throughput within a TCP streaming session causes receiver buffer starvation. When this happens, the video player stops playing video until it receives enough packets. Rebuffering rate is an important metric in video streaming as it reflects user experience watching videos. YouTube has a built-in mechanism to provide real-time monitoring of video playbacks. During a playback, the video player sends detailed information about user interactions to the server. The information includes the timestamps of all rebuffering events in each TCP connection. To quantify the user perceived performance of video streaming, we use rebuffering chance and rebuffering frequency suggested by previous works [18]. The rebuffering chance measures the probability of experiencing rebuffering events and is defined by percentage of flows that experience at least one rebuffering event. Rebuffering frequency measures how frequent the rebuffering events occur and is defined by r/T, where r is the number of rebuffering events and T is the duration of a flow.

Table 2 shows the average of rebuffering metrics in DC1 and DC2. DC2 users clearly have much worse experience than DC1 users. However, in both data centers the rebuffering frequency and rebuffering chance are similar between all four groups, suggesting Trickle has negligible impact on the streaming quality. Initially the results puzzled us as we expected Trickle to improve rebuffering by reducing burst drops. To explain the results, we studied the correlation of rebuffering and various network metrics. We found that the bandwidth deficit, the difference between the target streaming rate and the bandwidth, is the key factor for rebuffering. In both DC1 and DC2, among the flows that do not have sufficient bandwidth (positive deficit), 55% to 60% of them have experienced at least one rebuffering event. Another major factor is when a user requests a different resolution by starting a new connection.

4 Discussions and Related Work

Trickle is motivated by Alcock *et al.*'s work [2], which identified a YouTube burst drops problem in residential broadband and university networks. Further, Ashwin *et al.* showed that popular browsers also throttle the

video streaming in addition to server side throttling in YouTube and Netflix [6]. The bilateral throttling mechanisms sometimes result in packet bursts up to several MBs. Blanton *et al*. studied the correlation between burst size and losses in TCP [7]. They discovered that bursts less than 15 packets rarely experience loss but large (over 100) bursts nearly always do. Allman *et al*. evaluated several changes to mitigate bursts created by TCP [3].

We have also considered other solutions to rate limit video streaming. A similar idea that requires no kernel TCP change is to set the TCP send socket buffer size [19]. In the case of YouTube, the ustreamer TCP send buffer remains auto-tuned [20] during the startup phase in order to send data as fast as possible. Upon entering the throttling phase, the buffer usually is already larger than the intended clamp value. Setting a new send buffer size is not effective until the buffered amount drops below the new size, making it difficult to implement the throttling. Some previous work control the rate by dynamically adjusting the TCP receive window at the receiver or the gateway [15, 16, 21]. Instead, Trickle is server-based making it easier to deploy in a CDN. Another approach is TCP pacing [23], i.e., pacing *cwnd* amount of data over the RTT. While this may be the best TCP solution to suppress bursts, it is also more complex to implement. Moreover, studies have shown that Internet paths can absorb small amount of packet bursts [7,9]. Our goal is to reduce large burst drops caused by disruptions to the TCP self clocking. It is not to eliminate any possible burst completely.

5 Conclusions

The current throttling mechanism in YouTube sends bursts that cause losses and large queues. We presented Trickle, which removes these large bursts by doing rate-limiting in TCP. Trickle dynamically sets a maximum *cwnd* to control the streaming rate and strictly limit the maximum size of bursts. Through large-scale real user experiments, Trickle has effectively reduced the retransmissions by up to 50% in high bandwidth networks. It also reduces the average RTT by up to 28%.

References

[1] Sandvine global Internet report, Oct. 2011.

[2] ALCOCK, S., AND NELSON, R. Application flow control in YouTube video streams. *CCR 41* (April 2011), 24–30.

[3] ALLMAN, M., AND BLANTON, E. Notes on burst mitigation for transport protocols. *CCR* (Apr. 2005), 53–60.

[4] ALLMAN, M., EDDY, W. M., AND OSTERMANN, S. Estimating loss rates with TCP. *SIGMETRICS 31* (December 2003), 12–24.

[5] ALLMAN, M., PAXSON, V., AND BLANTON, E. TCP congestion control, September 2009. RFC 5681.

[6] ASHWIN, R., ARNAUD, L., YEON, L., DON, T., CHADI, B., AND WALID, D. Network characteristics of video streaming traffic. *CoNEXT* (Dec. 2011), 1–12.

[7] BLANTON, E., AND ALLMAN, M. On the impact of bursting on TCP performance. *PAM* (March 2005), 1–12.

[8] DUKKIPATI, N., MATHIS, M., CHENG, Y., AND GHOBADI, M. Proportional rate reduction for TCP. *IMC* (November 2011), 155–170.

[9] DUKKIPATI, N., REFICE, T., CHENG, Y., CHU, J., HERBERT, T., AGARWAL, A., JAIN, A., AND SUTIN, N. An argument for increasing TCP's initial congestion window. *CCR 40* (2010), 26–33.

[10] GHOBADI, M., CHENG, Y., JAIN, A., AND MATHIS, M. Trickle: Rate limiting YouTube video streaming. Tech. rep., Google Inc., January 2012. https://developers.google.com/speed/protocols/trickle-tech-report.pdf.

[11] GRINNEMO, K., AND BRUNSTROM, A. Impact of traffic load on SCTP failovers in SIGTRAN. *ICN* (2005), 774–783.

[12] HA, S., RHEE, I., AND XU, L. CUBIC: a new TCP-friendly high-speed TCP variant. *SIGOPS'08 42*, 64–74.

[13] HANDLEY, M., PADHYE, J., AND FLOYD, S. TCP congestion window validation, June 2000. RFC 6298.

[14] JIANG, H., AND DOVROLIS, C. Source-level IP packet bursts: causes and effects. *IMC* (October 2003), 301–306.

[15] KARANDIKAR, S., KALYANARAMAN, S., BAGAL, P., AND PACKER, B. TCP rate control. *CCR 30* (January 2000), 45–58.

[16] LOMBARDO, A., PANARELLO, C., AND SCHEMBRA, G. Applying active window management for jitter control and loss avoidance in video streaming over TCP connections. *IEEE Globecom* (December 2010), 1–6.

[17] MOHAMMAD, R., AND ANNA, B. On the Effectiveness of PR-SCTP in Networks with Competing Traffic. *ISCC* (2011), 898–905.

[18] MOK, R., CHAN, E., AND CHANG, R. Measuring the quality of experience of HTTP video streaming. *IFIP* (May 2011), 485–492.

[19] PRASAD, R., JAIN, M., AND DOVROLIS, C. Socket buffer auto-sizing for high-performance data transfers. *Journal of GRID computing 1*, 4 (2004), 361–376.

[20] SEMKE, J., MAHDAVI, J., AND MATHIS, M. Automatic TCP buffer tuning. *CCR 28* (October 1998), 315–323.

[21] SPRING, N., CHESIRE, M., BERRYMAN, M., SAHASRANAMAN, V., ANDERSON, T., AND BERSHAD, B. Receiver based management of low bandwidth access links. *INFOCOM* (March 2000), 245–254.

[22] TAHT, D., GETTYS, J., AND TURNER, S. The bufferbloat problem, 2011. http://www.bufferbloat.net/.

[23] ZHANG, L., SHENKER, S., AND CLARK, D. Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic. *CCR 21* (August 1991), 133–147.

Tolerating Overload Attacks Against Packet Capturing Systems

Antonis Papadogiannakis,[*] Michalis Polychronakis,[†] Evangelos P. Markatos[*]
[*]*FORTH-ICS,* [†]*Columbia University*
{papadog,markatos}@ics.forth.gr, mikepo@cs.columbia.edu

Abstract

Passive network monitoring applications such as intrusion detection systems are susceptible to overloads, which can be induced by traffic spikes or algorithmic singularities triggered by carefully crafted malicious packets. Under overload conditions, the system may consume all the available resources, dropping most of the monitored traffic until the overload condition is resolved. Unfortunately, such an awkward response to overloads may be easily capitalized by attackers who can intentionally overload the system to evade detection.

In this paper we propose Selective Packet Paging (SPP), a two-layer memory management design that gracefully responds to overload conditions by storing selected packets in secondary storage for later processing, while using randomization to avoid predictable evasion by sophisticated attackers. We describe the design and implementation of SPP within the widely used Libpcap packet capture library. Our evaluation shows that the detection accuracy of Snort on top of Libpcap is significantly reduced under algorithmic complexity and traffic overload attacks, while SPP makes it resistant to both algorithmic overloads and traffic bursts.

1 Introduction

Passive network monitoring systems have been increasingly used to improve the performance and security of our networks. These systems operate in unpredictable and sometimes hostile environments where transient traffic and malicious attackers may easily overload them up to the point where they cease to function correctly. Unfortunately, traditional packet capturing systems have not been designed for such hostile environments, and do not gracefully handle overload conditions. For example, when faced with overload conditions and full packet queues, most packet capturing systems start to discard all incoming packets for as long as the overload persists.

This naive approach to packet discarding has three major disadvantages: (i) it may drop packets that contain *important information*, such as an attack or a particular pattern; (ii) it can be exploited by attackers to *hide their attack* by flooding the system with bogus packets up to the point where the system overloads and starts discarding (i.e., not inspecting) most of the incoming packets [2, 15, 17]; (iii) it robs monitoring applications from the opportunity to *selectively discard the unimportant packets* in the traffic [9, 10, 12], and forward for processing and further inspection the *important* ones.

To cope with high traffic volumes, several techniques have been proposed for improving the performance of Network Intrusion Detection Systems (NIDSs) by accelerating the packet processing throughput [7, 14], or by balancing detection accuracy and resource requirements [3, 8]. However, even after carefully tuning a NIDS according to the monitored environment, it will still have to cope with inevitable traffic bursts or unpredictable algorithmic attacks.

To address these problems we propose *Selective Packet Paging (SPP)*, a novel approach for mitigating both traffic overloads and algorithmic attacks by exploiting the following two dimensions: (i) we introduce a *new level in the memory hierarchy* of packet capturing systems, a level which is able to store all packets during overload periods; and (ii) we propose a *randomized timeout algorithm* which is able to detect and isolate malicious packets that trigger algorithmic overload attacks.

The main contributions of this paper are: (i) we demonstrate that the root of packet discarding under overload in the current packet capturing system in Linux [11] is the poor design choices in memory management. (ii) we propose Selective Packet Paging, a novel two-layer memory management system that can store practically all network packets during overloads, and resolve algorithmic complexity attacks by removing from the critical path any malicious packets that slowdown a monitoring system; (iii) we implement Selective Packet Paging within the Libpcap packet capture library; (iv) we experimentally evaluate our approach using the Snort NIDS [16], and we show that it can sustain algorithmic attacks and traffic overloads without discarding any packets, while the traditional approach is forced to discard the largest percentage of the incoming packets and miss 100% of the attacks, and (v) we analytically evaluate the randomized timeout selection approach of SPP and show that the probability of detecting an algorithmic attack reaches certainty exponentially fast.

2 Selective Packet Paging

The main cause of packet loss during overloads is usually the limited number of packets that the Operating System's packet capturing subsystem can store in main memory. Thus, in case of traffic overloads or algorithmic attacks, the main memory fills up quickly and the rest of the incoming packets are just dropped. One obvious solution would be to increase the main memory available to the packet capturing subsystem. Unfortunately, main memory typically can not store more than a few seconds of network traffic for a high-speed link. Thus, an algorithmic attack or a network overload that lasts for more than a few seconds will eventually lead to packet drops.

In modern systems, the available disk storage is up to three orders of magnitude larger than the available storage in main memory. Thus, captured packets can be buffered on disk for several hours under overload conditions, instead of just a few seconds in main memory.

2.1 Multi-level Memory Management

In this paper we propose to break away from the single-level memory hierarchy traditionally used by packet capturing subsystems and employ a multi-level memory hierarchy consisting of at least two levels: a main memory and a secondary storage. Under normal circumstances captured packets are written in main memory. Under traffic overload or algorithmic attacks, when the main memory fills up, extra packets are written to secondary storage. Figure 1 presents the two-level memory hierarchy of our approach. As long as it is not full, newly arriving packets are written in the memory buffer. Upon filling up, newly arriving packets are stored in the second layer of the memory hierarchy, i.e., the disk buffer.

Note that while newly arriving packets are being written to disk, memory space is being freed up as monitoring applications continue to consume existing packets. In this case, we would like to be able to write newly arriving packets in main memory and thus avoid the disk access overheads. However, this choice implies that sequentially arriving packets may be written to different levels of the memory hierarchy, oscillating between main memory and disk. For this reason we use a Packet Receive Index which keeps the incoming packets strictly in FIFO order. To deliver packets in the correct order, we use one bit for each incoming packet in the Packet Receive Index, as shown in Figure 1. This bit indicates whether the packet was stored in main memory or on disk.

2.2 Randomized Timeout Intervals

Although multi-level memory management makes sure that no packets are lost during an overload, algorithmic attacks may force the CPU to spend most of its time on processing bogus attack packets that trigger an algorithmic overload—benign network packets will just keep accumulating on disk. Selective Packet Paging advocates that instead of blindly sending subsequent packets to secondary storage when the main memory is full, we should develop mechanisms to detect packets that trigger an algorithmic overload, weed them out, and send *them* to secondary storage for processing at a later point in time.

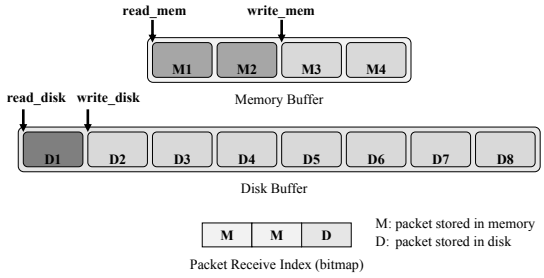


Figure 1: **A snapshot of Packet Paging for buffering packets to memory and disk.** The Packet Receive Index indicates that the first two packets are stored in the Memory Buffer, while the third packet is in the Disk Buffer.

To detect packets that trigger algorithmic overload attacks we could use a timeout counter: when a new packet arrives, the counter is initialized to a timeout value larger than the processing time of benign packets. If the timeout expires while the application is still processing the same packet, then this packet is considered suspicious. Unfortunately, setting timeouts on each and every packet imposes a prohibitively large processing overhead.

To reduce this overhead, the counter can be set at *periodic* intervals: if the packet being processed during the interval expiration is the same packet that was being processed at the time the counter was set, then SPP considers this packet as suspicious. As a result, the packet, along with all subsequent packets from the same network flow, will be buffered to disk. Although setting the timeout at periodic intervals has the potential to reduce the processing overhead, choosing an appropriate timeout can be challenging: a very large value may miss a lot of attack packets, while a very small value may impose a large processing overhead. To make matters worse, a single predefined timeout value (or a deterministic sequence of timeout values) could theoretically be evaded by a sophisticated attacker who manages to send all attack packets between successive timeouts.

To solve this problem, SPP uses a *randomized* timeout interval. Instead of choosing a predefined constant timeout, SPP selects a random timeout uniformly distributed in the interval $[low, high]$. Choosing a large value for $high$ reduces the average timeout overhead, while choosing a small value for low makes detection of algorithmic attacks easier. Indeed, to make sure they avoid detection, attackers should only send attack packets that impose a processing delay of no more than low seconds. Therefore, a small low value forces a (what used to be) sophisticated algorithmic attack to degenerate into a brute force Denial of Service attack consisting of a torrent of attack packets, which can be easily detected and filtered out.

3 Implementation

We have implemented Selective Packet Paging within the popular packet capturing library Libpcap [11], so that existing network monitoring applications can benefit from SPP without any code modifications. In our prototype implementation we use three separate threads: (i) the *packet capturing and storing thread*, which receives packets from the NIC and stores them to memory or disk; (ii) the *packet processing* thread, which finds the next packet through the Packet Receive Index, and calls the callback function for processing each packet; and (iii) the *disk I/O thread*, which handles all communication with the secondary storage. We give higher priority to the packet capturing thread over the packet processing thread to ensure that all packets will be stored during overloads. To optimize disk throughput, the disk I/O thread transfers packets between main memory and disk in batches. Moreover, to avoid delays from blocking read operations, the disk I/O thread prefetches the next batch of packets from disk to a memory cache.

The processing thread keeps a counter of the processed packets. When the timer expires, it checks how many packets have been processed from the previous timer expiration. If the number of processed packets remains the same, then the current packet delays the system for an unreasonably long time. Thus, the packet is evicted and buffered to disk, while its flow and source IP address are marked as suspicious. Packets belonging to suspicious flows are written to disk as low priority packets. If there are no normal priority packets in the queues, then a low priority packet is processed. The next timer interval is scheduled to a random time between the low and $high$ limits. The timer expires based on the time passed while only the current process is executing, so SPP is not affected by external background activities. To avoid false positives, a proper value for the low limit should be used. Then, only packets with significant processing delays will be detected as suspicious. But even in case of false positives, packets will not be dropped. They will be processed when the system has the available resources.

4 Analytical Evaluation

Using a random timeout uniformly distributed in the range $[low, high]$, SPP makes it difficult for attackers to evade detection, while keeping the timeout overhead reasonably low. Since, however, the timeout is a random variable, it is theoretically possible even for an attack packet that triggers a long algorithmic attack to evade detection. This is especially true if the timeout interval chosen while the attack packet is being processed is relatively large. In this section we show that although it is theoretically possible for one attack packet to evade detection, it is very unlikely that several attack packets will go undetected. An attacker who wants to sustain an algorithmic attack has to send several attack packets, and it is improbable that none of them will be detected.

To simplify our analysis, we initially assume that there are only attack packets, that each attack packet is being analyzed for a constant interval of d µs, and $low < d$. Selective Packet Paging can detect an attack if two successive timeouts expire within the same interval for the same attack packet. The first timeout expires at time t_1, which will fall within an interval i of an attack packet. Thus, $i \times d < t_1 < (i+1) \times d$. The probability that the second timeout t_2 will also fall within the interval i is:

$$P(t_2 < (i+1) \times d) = \frac{d - t_1 - low}{high - low} \quad (1)$$

since there are $high - low$ possible choices for a timeout but only $d - t_1 - low$ accepted choices so that the second timeout expires within the interval i. In the unfortunate for the attacker case that t_1 falls in the beginning of the interval i, there are $d - low$ accepted choices for t_2. In case t_1 falls in the position $(i+1) \times d - low - 1$ of the interval i, there is only one accepted choice for t_2: the low timeout value. On average, there are $(d - low)/2$ accepted choices for t_2 in case t_1 falls within the first $(d - low)$ values of the interval i. If t_1 falls in the last low values of the interval i, there is no accepted choice for t_2. Overall, the probability for detection with two timeouts in the same interval is:

$$P(det) = \frac{(d - low)^2}{2 \times d \times (high - low)} \quad (2)$$

since the possible choices for two timeouts are $(high - low) \times (high - low)$, the accepted choices for the first timeout are $(high - low)$, and the accepted choices for the second timeout are $(d - low)/2 \times (d - low)/d$.

The probability of not detecting an attack after N timeouts have expired is $(1 - P(det))^N$, and thus the probability of detecting the attack after N timeouts is $1 - (1 - P(det))^N$: we see that the detection probability approaches 1 very fast as N increases. Also, the detection probability from Equation 2 implies that, on average, SPP will need $T = 1/P(det) + 1$ timeouts to detect the attack. This number corresponds on average to $T \times (high - low)/(2 \times d)$ attack packets and $T \times (high - low)/2$ µs.

The outcomes of our analysis are also valid in case that the attack packets induce variable delays with an average delay of d µs. In a more realistic scenario there will be both benign and attack packets, so that attack packets will be a percentage a of the total packets, with $0 < a < 1$. The average processing time for a benign packet is t µs, and we expect that $t < d$. In this case the detection probability from Equation 2 is:

$$P(det) = \frac{a \times d}{d + t} \times \frac{(d - low)^2}{2 \times d \times (high - low)} \quad (3)$$

since the probability of the first timeout to expire within an interval of an attack packet is $a \times d/(d + t)$.

To validate our analysis we compare its results with a simulation-based evaluation. Figure 2 presents the detection time as a function of the processing time for each

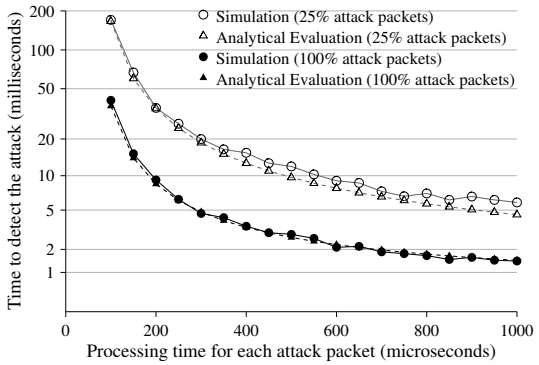


Figure 2: **Detection time as a function of the processing time of attack packets.**

attack packet for two attack scenarios: i) when all packets are attack packets, and ii) when the percentage of attack packets is 25%. The processing time t of each benign packet is uniformly distributed between 1 and 30 µs, while the processing time d of each attack packet is constant for each simulation. We vary d from 100 to 1000 µs to examine how the detection time is affected. The randomized timeout for SPP is randomly chosen between low=50 and $high$=1000 µs. When two successive timeouts expire during the processing interval of the same attack packet, the experiment ends and the detection time is recorded. Each experiment was repeated a million times.

For the analytical evaluation we used the probability from Equation 3 to compute the number of timeouts T needed for the detection:

$$T = 1/P(det) + 1 = \frac{2 \times d \times (high - low) \times (d + t)}{(d - low)^2 \times a \times d} + 1 \quad (4)$$

The average detection time is $T \times (high - low)/2$ µs.

In Figure 2 we can see that simulation results are very close to the expected results based on our analysis. We observe that SPP with the randomized timeout can detect even attacks with very small delays within just a few milliseconds. For instance, when the processing time of an attack packet is 200 µs, SPP detects the attack within the first 10 ms in case all packets belong to this attack. In a more conservative attack, where only 25% of the total packets impose 100 µs processing time, SPP needs about 170 ms to detect it. However, such a conservative attack for a period of a few milliseconds will not affect significantly the system. More aggressive attacks are detected by SPP within less than 2 ms.

5 Experimental Evaluation

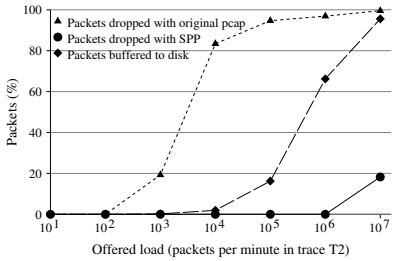
Our testbed consists of two PCs interconnected through a 10GbE switch. The first is used for traffic generation, which is achieved by replaying real network traffic at different rates. The second (NIDS PC) is equipped with two quad-core Intel Xeon 2.00 GHz CPU with 6 MB L2 cache, 4 GB RAM, and a 10GbE network interface. Beyond the system disk, the NIDS PC has four 750 GB 7200 RPM SATA disks organized in RAID 0 (totaling 3 TB of secondary storage for SPP), which can sustain a 3 Gbit/s read and 1.8 Gbit/s write throughput. The size of the memory buffer for storing packets is set to 1 GB in all cases. We use Snort v2.8.3.2 [16] with the latest official Sourcefire VRT rule set, containing 9276 rules.

For our evaluation we use three traces. As real background traffic, we replay a one-hour full payload trace (named `T1`) captured at the access link that connects a large university campus to the Internet. The trace contains 58,714,906 packets, corresponding to 1,493,032 flows, with an average traffic rate of 110 Mbit/s. The second trace (`T2`) is used to trigger an algorithmic overload in Snort using crafted packets that exploit the backtracking vulnerability of a regular expression used in a particular rule. The third trace (`T3`) contains 120 real attacks that are detected by Snort using the default rule set, resulting to 276 alerts from 14 different rules. We replay this trace continuously and measure the alerts that Snort was able to detect with the original Libpcap and SPP.

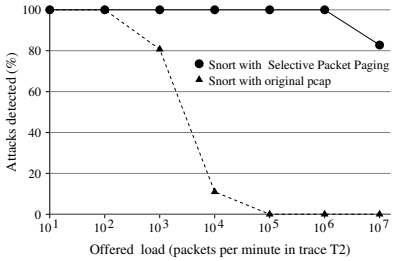
5.1 Algorithmic Complexity Attack

In this experiment we perform an algorithmic complexity attack against Snort, which uses the PCRE library [5] for regular expression matching, as described by Smith et al. [17]. For a given input string, PCRE iteratively explores paths in its internal tree-like structure until it finds a matching state. If it fails to find a match, it backtracks and tries another path until all paths have been explored. As the number of backtracks increases, more time is spent on matching, and overall performance decreases. The attack we use targets the Snort rule 2682, which detects exploitation attempts of a known vulnerability that allows e-mail attachment execution. We created 1500-byte packets belonging to an established connection destined to port 25 (trace `T2`). When processed, each crafted packet results to a processing time about 1360 times slower compared to the average time that Snort spends for processing SMTP packets in trace `T1`.

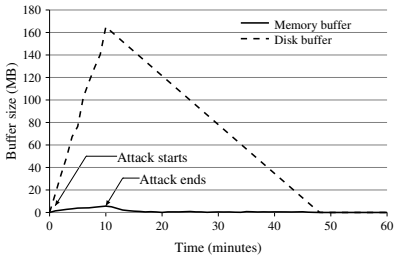
We set out to explore what is the packet loss of the Libpcap and SPP during this algorithmic overload attack. While we replay the background traffic and the actual attacks (traces `T1` and `T3`) at low rates, we also replay the `T2` trace at a variable rate, from 10 crafted packets up to 10^6 crafted packets/min. Figure 3(a) shows the percentage of dropped packets when Snort runs on top of the original Libpcap and on top of SPP. We observe that when the traffic load reaches a mere 10^3 packets/min, Libpcap starts losing packets, and when the load exceeds 10^4 it loses more than 80% of the packets. On the contrary, at these loads, SPP loses no packets and manages to store them to disk. Figure 3(a) also shows that the packets buffered to disk by SPP are fewer than those dropped by Libpcap at similar rates. This is because by identifying and weeding out algorithmic attack packets, SPP frees more CPU cycles for processing ordinary packets.



(a) Percentage of dropped packets and packets buffered to disk as a function of the offered load.

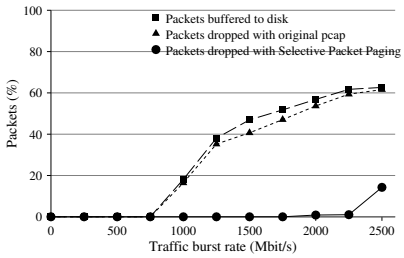


(b) Percentage of detected attacks as a function of the offered load.

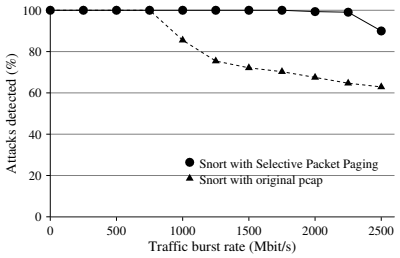


(c) Size of memory and disk buffers over a 60-minute time period when sending 10,000 crafted packets/minute for the first 10 minutes.

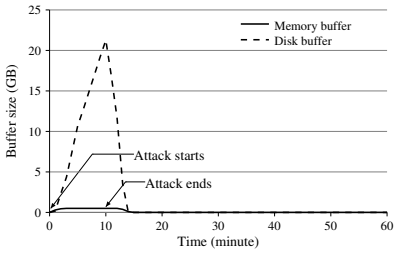
Figure 3: Performance of SPP and original Libpcap under an algorithmic complexity attack.



(a) Dropped packets and packets buffered to disk as a function of the traffic burst rate.



(b) Detected attacks as a function of the traffic burst rate for 30-second bursts.



(c) Size of memory and disk buffers over a 60-minute time period when sending 30-second traffic bursts of 1.5 Gbit/s for the first 10 minutes.

Figure 4: Performance of SPP and original Libpcap in case of 30-second traffic bursts.

Packet loss is directly translated to undetected attacks. Figure 3(b) shows the percentage of attacks detected by the two systems. We see that Snort on top of Libpcap starts missing attacks when the traffic load exceeds 10^2 packets/min, and misses all attacks as the load reaches 10^5 packets/min. At the same rates, Snort on top of SPP detects *all* attacks, as all packets are stored to secondary storage and are eventually inspected, and does not miss any attack for rates up to 10^6 packets/min. That is, an attacker needs to send about 10^7 packets/min to reduce the probability of being detected just by 17%. In this extreme case, SPP was not able to store all incoming packets to disk due to the high traffic rate. Compared to the original Libpcap, SPP can handle 10,000 times more crafted packets, offering significant tolerance against highly efficient algorithmic complexity attacks.

To measure the time that the system needs to recover from overload, we replayed traces `T1` and `T3` at low rates for 60 minutes, and replayed the `T2` trace at a rate of 10^4 packets/min for the first 10 minutes of the experiment. Figure 3(c) presents the size of memory buffer and disk buffer over time. We observe that with SPP the size of the memory buffer was always less than 6 MB, for the whole 60-minute period. The attack packets (and their associated flows) identified by SPP were sent to disk. Indeed, to accommodate the attack packets, the disk buffer size increased from 16.23 MB (at minute 1) to 165 MB (at minute 10, which was the highest point of the attack), and then slowly decreased back to zero at minute 48.

5.2 Traffic Overload

In this experiment we explore how Snort on top of SPP and the original Libpcap responds to traffic bursts. We replay trace `T1` at its original rate as background traffic and trace `T3`, containing the 120 real attacks, at 1 Mbit/s for the entire duration of the experiment. At each minute we send a traffic burst that lasts for 30 seconds using traffic from `T1`. The peak rate of the burst is varied from 1 Gbit/s up to 2.5 Gbit/s, to evaluate how burst intensity may influence SPP. Each experiment lasts 10 minutes.

Figures 4(a) and 4(b) present the percentage of dropped packets and detected attacks as a function of the rate of the 30-second traffic bursts. We observe that Libpcap starts dropping packets when the traffic bursts are around 1 Gbit/s, resulting in about 17% undetected attacks. When the bursts reach a rate of 2 Gbit/s, 53% of the packets are dropped and 32.5% of the attacks are missed. On the other hand, Snort with SPP drops no packets and misses no attacks even at rates as high as 2 Gbit/s. Although our disk system writes packets with a throughput of 1.8 Gbit/s, the two-level memory hierarchy allows processing of 2 Gbit/s traffic without packet loss. Only when the burst rates exceed 2.25 Gbit/s the secondary storage is not able to keep up with network traffic and SPP starts losing packets.

Figure 4(c) shows the size of memory and disk buffers when sending 30-second traffic bursts with a rate of 1.5 Gbit/s for 10 minutes, and continue sending only background traffic for another 50 minutes. The memory

buffer remains full at 500 MB for the first 12 minutes, while the disk buffer fills up continuously during the first 10 minutes, all the way up to 21.3 GB. From minute 11 to minute 13, the disk buffer size is reduced from 21.3 to 3.5 GB, as the system's resources are sufficient to process the excessive packets buffered during the traffic bursts. Thus, in the 14th minute, both memory and disk buffers are empty, so the system has fully recovered from the traffic overload attack. Compared with the algorithmic complexity attack, the system recovers faster from this traffic overload, (within just four minutes) because packets are not maliciously crafted to further slowdown Snort.

6 Related Work

To cope with high traffic volumes, several research approaches propose to distribute the load across multiple computers instead of using a single sensor [7], or utilize multi-core processors for parallel inspection [4, 14]. Other approaches propose to dynamically reconfigure a NIDS based on the run-time conditions [3,8], or use load shedding techniques to defend against overloads [1, 13]. Recent works deal with high traffic volumes by applying a per-flow cutoff to selectively discard most of the traffic and focus on the beginning of each connection when the system is under load [9,10,12]. Unfortunately, overloads are still possible in all these systems, especially in case of algorithmic attacks [15, 17].

Smith et al. [17] propose memoization as an algorithmic solution to prevent overload attacks targeting backtracking-based algorithms. Crosby and Wallach [2] present an algorithmic complexity attack that exploits deficiencies of common data structures, and propose new hashing techniques which sacrifice average case performance for worst case performance. Khan and Traore [6] propose a model to detect algorithmic complexity attacks based on historical information of execution time and input characteristics, using regression analysis.

7 Conclusion

We presented Selective Packet Paging, a two-level memory management approach that buffers (otherwise dropped) packets to tolerate algorithmic complexity attacks and traffic overloads for network monitoring and security applications. Empowered with a randomized timeout, SPP can detect and isolate algorithmic attack packets, enabling the CPU to be used for more useful purposes. We have implemented SPP within the popular Libpcap packet capture library, so that existing applications can use it without any code modifications. Our experimental evaluation shows that NIDS, such as Snort, are vulnerable to both algorithmic complexity and traffic overload evasion attacks. Using SPP, a NIDS can handle both algorithmic and traffic overload conditions.

We believe that as network monitoring applications get more complicated, they will be increasingly vulnerable to algorithmic and traffic overload attacks. SPP offers a memory management approach and a dynamic overload detection technique that provide a seamless solution to this problem without requiring any changes to the monitoring applications themselves.

Acknowledgments

We would like to thank our shepherd Samuel T. King and the anonymous reviewers for their valuable feedback. This work was supported in part by the FP7-PEOPLE-2009-IOF project MALCODE and the FP7 project SysSec, funded by the European Commission under Grant Agreements No. 254116 and No. 257007.

References

[1] P. Barlet-Ros, G. Iannaccone, J. Sanjuàs-Cuxart, D. Amores-López, and J. Solé-Pareta. Load shedding in network monitoring applications. In *Proc. of the USENIX Annual Technical Conf. (ATC)*, 2007.

[2] S. A. Crosby and D. S. Wallach. Denial of service via algorithmic complexity attacks. In *Proc. of the 12th Conf. on USENIX Security Symp.*, pages 3–3, 2003.

[3] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational experiences with high-volume network intrusion detection. In *Proc. of the 11th ACM Conf. on Computer and communications security (CCS)*, pages 2–11, 2004.

[4] F. Fusco and L. Deri. High speed network traffic analysis with commodity multi-core systems. In *Proc. of the 10th annual Conf. on Internet measurement (IMC)*, pages 218–224, 2010.

[5] P. Hazel. Pcre: Perl compatible regular expressions. http://www.pcre.org.

[6] S. Khan and I. Traore. A prevention model for algorithmic complexity attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment, Second Intern. Conf., (DIMVA)*, pages 160–173, 2005.

[7] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer. Stateful intrusion detection for high-speed networks. In *Proc. of the IEEE Symp. on Security and Privacy*, pages 285– 294, 2002.

[8] W. Lee, J. B. D. Cabrera, A. Thomas, N. Balwalli, S. Saluja, and Y. Zhang. Performance adaptation in real-time intrusion detection systems. In *Proc. of the 5th International Symp. on Recent Advances in Intrusion Detection (RAID)*, pages 252–273, 2002.

[9] T. Limmer and F. Dressler. Improving the Performance of Intrusion Detection using Dialog-based Payload Aggregation. In *14th IEEE Global Internet Symp. (GI)*, pages 833–838, 2011.

[10] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider. Enriching network security analysis with time travel. In *Proc. of the 2008 Conf. on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM)*, pages 183–194, 2008.

[11] S. McCanne, C. Leres, and V. Jacobson. libpcap. Lawrence Berkeley Lab., Berkeley, CA. (http://www.tcpdump.org/).

[12] A. Papadogiannakis, M. Polychronakis, and E. P. Markatos. Improving the accuracy of network intrusion detection systems under load using selective packet discarding. In *Proc. of the Third European Workshop on System Security (EUROSEC)*, pages 15–21, 2010.

[13] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.

[14] V. Paxson, R. Sommer, and N. Weaver. An architecture for exploiting multi-core processors to parallelize network intrusion prevention. In *Proc. of the IEEE Sarnoff Symp.*, 2007.

[15] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., 1998.

[16] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proc. of the 1999 USENIX LISA Systems Administration Conf.*, 1999.

[17] R. Smith, C. Estan, and S. Jha. Backtracking algorithmic complexity attacks against a nids. In *Proc. of the Annual Computer Security Applications Conf. (ACSAC)*, 2006.

Enforcing Murphy's Law for Advance Identification of Run-time Failures*

Zach Miller
zmiller@cs.wisc.edu
University of Wisconsin–Madison

Todd Tannenbaum
tannenba@cs.wisc.edu
University of Wisconsin–Madison

Ben Liblit
liblit@cs.wisc.edu
University of Wisconsin–Madison

Abstract

Applications do not typically view the kernel as a source of bad input. However, the kernel can behave in unusual (yet permissible) ways for which applications are badly unprepared. We present *Murphy*, a language-agnostic tool that helps developers discover and isolate run-time failures in their programs by simulating difficult-to-reproduce but completely-legitimate interactions between the application and the kernel. Murphy makes it easy to enable or disable sets of kernel interactions, called *gremlins*, so developers can focus on the failure scenarios that are important to them. Gremlins are implemented using the `ptrace` interface, intercepting and potentially modifying an application's system call invocation while requiring no invasive changes to the host machine.

We show how to use Murphy in a variety of modes to find different classes of errors, present examples of the kernel interactions that are tested, and explain how to apply delta debugging techniques to isolate the code causing the failure. While our primary goal was the development of a tool to assist in new software development, we successfully demonstrate that Murphy also has the capability to find bugs in hardened, widely-deployed software.

1 Introduction

1.1 Motivation

Despite extensive in-house regression testing, buggy software is still released for a variety of reasons including incomplete test coverage, unexpected user inputs, and different run-time environments. Software developers want to systematically discover, identify, and fix application run-time failures before they affect users in the field. One challenge towards accomplishing this lofty goal is nondeterministic behavior at the level between the application and the kernel. A typical application makes thousands of calls into the kernel, and most of the time these calls respond in a repeatable manner. However, under certain run-time environment conditions, system calls into the kernel that typically succeed may return with legitimate but unexpected values.

A simple example is the `write()` system call: it usually succeeds when given valid input parameters, but fails if the disk is full. Does a given program behave in an acceptable and predictable manner in the event of a full disk? Often development teams only learn the answer when users report failures in the field. Another example is the `read()` system call, which can legitimately return fewer bytes than requested by the caller. This may happen if an interrupt occurs or if a slow device does not have all requested data immediately available. Do programs always check the number of bytes returned by a `read()` and react appropriately?

Complicating the situation is the fact that environmental conditions which bring about unexpected return values from the kernel are often hard to replicate in a typical automated testing environment. For instance, how should a regression test suite validate proper behavior in the event of a full disk? Actually filling the disk to capacity causes problems for other processes on the machine. Mounting a loopback device volume requires superuser privileges [12]. Even creating a virtual machine with a full disk may not solve the problem, as this could cause faults in the test harness itself. Other environmental conditions can be even more challenging to reproduce. The consequence is that developers fail to perform continuous integration testing under these conditions.

Across many imperfect human endeavors, Murphy's Law pessimistically predicts that "**If anything can go wrong, it will.**" Unfortunately, this does not apply when testing software. Testing would find more bugs sooner if Murphy's Law were more strictly enforced.

*Supported in part by DoE contract DE-SC0002153, LLNL contract B580360, and NSF grant CCF-0953478. Opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of NSF or other institutions.

1.2 Approach

Given the observation that a program ultimately interacts with its environment via the kernel interface, we offer a tool, called *Murphy*, to serve as an interposition agent between the application being tested and the kernel interface. Interposing at the kernel interface allows us to simulate a wide variety of environmental events. We allow enabling and disabling different sets of system call transformations, or *gremlins*, so developers can focus on the failure scenarios that are important to them. For example, when the application requests bytes from a file descriptor, the *readone* gremlin rewrites the system call to ask for and return one byte at a time.

Beyond the gremlins themselves, Murphy offers several additional mechanisms to steer its behavior. A flexible activation policy language lets developers focus gremlin activity based on the call location, values of actual arguments to the call, and various other run-time properties. A replayable gremlin activation log allows deterministic reproduction of failures and iterative root-cause analysis via delta debugging [17]. The Murphy run-time API lets programs under test dynamically steer Murphy's actions based on the program's own internal state, further supporting automated testing and debugging.

The remainder of this paper is organized as follows. Section 2 describes the architecture of Murphy, including example gremlins and run-time steering mechanisms. Section 3 provides our results running Murphy, and related work is presented in Section 4. We conclude and suggest future work in Section 5.

2 Architecture and Implementation

Our descriptions here are necessarily brief; additional details appear in a companion technical report [10].

2.1 System Call Interposition

We use a customized version of the Parrot Virtual File System tool [15] as the basis for our interposition mechanism. Parrot handles core tasks such as intercepting I/O-related system calls, decoding arguments, and replacing selected calls with new functionality. All of these actions are performed in user-space with no kernel modifications or special administrative privileges. Most uses of Parrot concern I/O virtualization for large-scale, distributed systems. We use Parrot here to simplify building gremlins.

2.1.1 Use of `ptrace`

Our implementation uses the `ptrace` interface to optionally modify interactions with the kernel, essentially acting as a "system interface interposition agent." [7] When the application under test invokes a system call, the kernel suspends that process and passes control to Murphy, which intercepts and inspects the call. At this point, Murphy may decide to tamper with the program's execution by using the `ptrace` mechanism to peek (read) or poke (write) bytes into the traced program's address space. Depending upon the system call trapped and which gremlins are configured to be active, Murphy will either

1. pass the system call to the kernel and then pass the response back to the application without any modification of the input or output arguments;
2. immediately return a failure to the application without actually passing the request to the kernel; or
3. modify the input arguments to the system call before passing the call to the kernel, and pass the actual response back to the application.

Murphy is able to track and trace entire process families by trapping `fork()`, `clone()`, `getppid()` and others, and forwarding signals and process exit codes.

2.1.2 Trade-offs of `ptrace` Interposition

Our system call interposition approach has pros and cons. There is great ubiquity by trapping via `ptrace`, and great flexibility by interposing in user-space. One major benefit is being language agnostic: Murphy works with applications written in any language, including increasingly popular managed languages such as Python and Java. No source code is required, and environmental failures can be simulated without root privileges and without impacting other processes on the system not targeted for testing. Because Murphy supports tracing entire process trees, it is possible to test software stacks (such as LAMP) consisting of many different programs written in different languages, in addition to programs that are statically linked and/or linked with C run-times other than glibc.

This approach also has challenges. The Linux system call interface does not necessarily correspond neatly to application actions. For example, all of the network socket calls are multiplexed into one (complicated) system call. Similarly, the mapping between thread creation and coordination as familiarly described by the POSIX threads API manifests itself via a strange brew of `clone()` and `futex()` system calls. When developing new gremlins, figuring out how these APIs map onto the system call interface can be a time-consuming exercise. Furthermore, recent versions of the Linux kernel introduced *vsyscall* and *vDSO* mechanisms to accelerate system calls that do not require any real level of privilege to run, such as `gettimeofday()` [1]. Calls that use these mechanisms do not cross the user/kernel boundary and therefore are invisible to Murphy.

2.2 Gremlins

We implement several example gremlins within this general framework. These include gremlins that immediately return legitimate error codes such as errno `EINTR` or errno `EAGAIN`; gremlins that modify `read()` and `write()` to simulate interrupted I/O; gremlins that introduce different amounts of latency; and special-purpose gremlins that span multiple system calls, for example simulating a full disk partition by returning errno `ENOSPC` in situations where that would make sense. In general, gremlins can further be divided into two categories: *halting* and *non-halting*. Halting gremlins typically prevent the application from making any further progress, such as *enospc* that simulates a full disk. When enabling halting gremlins, a developer can test that an application does not simply crash or abort, but instead correctly handles the situation by shutting down in an acceptable manner and reporting the error to the end-user. On the other hand, non-halting gremlins such as *readone* (causes `read()` to return one byte at a time) should not typically cause program failure. If a program's regression test suite passes with no gremlins, it should continue to pass with any non-halting gremlins activated.

Gremlins require defined composition and precedence rules. For example, both the *enospc* and the *writeone* gremlins tamper with the `write()` system call. If two or more gremlins trap the same system call, can their behaviors be combined, and if not, which one should have priority? In our current implementation, composition and precedence rules are hard-coded into Murphy.

2.2.1 Challenges to Writing Realistic Gremlins

Implementation of a single gremlin may require trapping multiple system calls. For instance, consider the *enospc* gremlin. Trapping just `write()` is not sufficient to simulate a full disk. `open()`, `mknod()`, `mkdir()`, `rename()` and over a dozen other system calls could fail due to a full disk. Murphy traps all of these.

Other gremlins may need to trap and record data from multiple system calls in order to correctly reconstruct kernel state and keep interactions legitimate. For example, consider the *cwdlongpath* gremlin that simulates executing the program with the current directory set to a very long path. At first blush, this sounds simple: just trap the `getcwd()` system call and return errno `ERANGE` if the size of the caller's buffer is smaller than the maximum allowed POSIX path length. But what if the program explicitly does a `chdir()` to `/usr`, and then invokes `getcwd()`? If `/usr` is not a symbolic link, the caller may safely assume that a smaller buffer is sufficient.

Another example is our desire for gremlins which operate on file descriptors to be conditionally activated based on the fully qualified path name referenced by the descriptor. To accomplish this, Murphy always traps `open()` to maintain mappings from file descriptors to file names. At later calls, these mappings allow "decoding" file descriptors so that they can be made available as file names for use with gremlin conditional activation (see Section 2.3). Argument decoding requires extra care for gremlins that operate on multiple system calls, as the meanings of arguments vary from one call to another. Finally, to make this useful in practice (for example, simulating `/tmp` being full), Murphy also needs to store path names that are fully qualified and canonicalized, meaning Murphy needs to track the current working directory, resolve relative paths, and expand symbolic links.

2.3 Use, Configuration, and Run-time API

To use Murphy, a developer simply invokes it with the name of the program to debug as a command-line argument. Optional command-line switches can specify the location of a configuration file and/or request the creation or replay of a gremlin activation log (see Section 2.4).

Each gremlin can be independently configured to be active, inactive, or conditionally active using a text-based configuration file. Activation conditions can be expressed in the ClassAd declarative policy language [14], providing great flexibility. A condition can be as simple as a random activation probability or can be more complex such as "activate when the file descriptor passed to this call corresponds to a file name that matches `lib*.so`."

A run-time API complements and extends static configuration. By calling into this API, the application under test can set arbitrary metadata, which is included in the gremlin activation log. Metadata might include source location information or relevant program state variables. Additional API functions allow the configuration described above to be modified dynamically for fine-grained, program-directed control over gremlin activation. Lastly, API functions allow the program to detach from Murphy and either suspend execution or immediately attach a debugger to the program under test. This helps the programmer follow their code into an area where they suspect it misbehaves. Taken together, the facilities offered by Murphy's run-time API help bridge the gap between an observed failure and the real root cause.

2.4 Reproduction of Failures

Reliably reproducing failures is essential to software testing and debugging. If Murphy is to assist developers beyond just alerting them to the existence of a bug, it must be able to reproduce the problem on demand. Note that even if a program's system call profile is deterministic, the interleaving of system calls across multiple processes is decidedly non-deterministic. In order to reproduce

gremlin-induced failures in multi-process code, we minimize non-deterministic behavior as follows:

1. Each gremlin has a separate pseudo-random number generator (PRNG) seed and state. Invoking the *readone* gremlin any number of times does not affect the PRNG for the *writeone* gremlin.
2. Multiple invocations of Murphy yield the same sequence of pseudo-random numbers.
3. For each process spawned by the application under test, Murphy maintains distinct system call statistics, gremlin states, PRNG state, and metadata.
4. Because the process ID (pid) assigned by the operating system changes during each re-run, Murphy assigns each newly spawned processes a virtual, monotonically increasing pid, or *vpid*. System call activity by this process is tracked using the tuple (pid, vpid).

Murphy can log an event whenever a gremlin modifies a system call. This log, called the *gremlin activation log*, contains a record indexed by the tuple (gremlin name, vpid) with the following fields: (1) how many times this particular gremlin was consulted to see if it wanted to modify the system call, (2) how many times Murphy has actually modified the system call, (3) the total count of all system system calls invoked by this vpid, and (4) the current value of user-supplied metadata for this process. Because this log uses the virtualized pid, and keeps track of the various system call statistics per vpid, successive runs of Murphy tracing the same program yield the same results, provided the program itself is deterministic.

Murphy can be instructed to replay the gremlin activation log while executing the program again, which produces the same results for deterministic programs. Murphy prints a warning if the count of total system calls for a given process does not match the log when a gremlin activation is replayed, letting the user know that things are not replaying identically. However, this is not fatal. In fact, it must be allowed later when minimizing the replay log (Section 2.5): removing certain gremlin invocations (such as *readone*) can affect how many subsequent system calls (such as `read()`) occur.

2.5 Fixing Failures

One disadvantage of interposing at the system-call level is a disconnect between these calls and the application developer's view of the operations being performed. This disconnect could create an understanding gap when it comes time for the developer to localize and fix errant behavior discovered by Murphy. While we assert that the mere existence of a tool that can discover such errors on a multi-process and possibly multi-language application is of value, we also support an automated strategy to help bridge this gap.

The first step is to use delta debugging [17] to shrink the failure-inducing gremlin activation log, thereby isolating just a few system calls that need to be manipulated to reproduce the failure. The second step uses Murphy to replay the minimized gremlin activation log, but now configured to suspend and detach from the application immediately upon replaying the last event in the log.

Delta debugging makes it easy for the programmer to focus their attention on the important system calls that behaved differently under Murphy. However, while very effective at minimizing gremlin activity, this does not completely bridge the gap between kernel interactions and source code. Thus, suspending after the last event leaves the program in a state where things are just about to go wrong. The user can attach with a debugger and directly observe the program's response to the manipulated system calls. In our experiments, this often results in a stack trace that pinpoints the exact line of buggy code.

3 Experimental Results

We applied Murphy to a variety of heavily-used open source packages. We ran the regression test suites of these packages primarily with non-halting gremlins enabled, and considered failed tests as bug candidates. We applied delta debugging per Section 2.5 to narrow down the code to be inspected. Often the activation log shrank to just a single system call, correlated with exactly one line of code. For example, we found a Perl interpreter bug by starting with an activation log containing 114,019 interleaved read and write system calls; delta debugging reduced this to just one vulnerable call that sufficed to cause the failure.

3.1 Ability to Detect and Pinpoint Bugs

Practically everything we tested failed with the *eagain* and *eintr* gremlins enabled. We could not run a single regression test suite with these gremlins active, as many test harnesses rely on tools like *make* that failed under the influence of these gremlins. The man pages for various system calls clearly document that they may return errnos `EAGAIN` or `EINTR`. Yet it seems that almost nothing actually checks for them. We also decided to forgo systematic testing with the latency-introducing gremlins given the limited time available for experimentation, because these obviously make the test suites run much slower.

Given the above, we focused our efforts primarily on testing with the *readone* and *writeone* gremlins. We found that even widely-used software failed to check for (or retry after) short reads or writes. Bash and Perl failed with short writes, while the widely-used OpenSSL library failed with short reads. The ubiquitous glibc also failed with short

reads: the Linux dynamic loader failed if it could not read an executable or shared library's ELF header in one `read()`. Even the trivial `/bin/true` program failed in this manner. This is a sobering sign that the problems Murphy targets are truly endemic, affecting even the most basic functionality of the system.

Problems were by no means limited to C code, or even to compiled code in general: short writes caused failures in both the Perl and Python regression test suites. The Perl and Python interpreters propagated Murphy-induced unusual behavior up into scripts. This is consistent with both interpreters' documented behavior, but the scripts themselves were unprepared for the consequences. Across a variety of application domains and languages, the methodology discussed in Section 2.5 allowed us to quickly and easily pinpoint each bug in the source code. We have reported some bugs to upstream developers and expect to report more in the future.[1]

3.2 Performance

Instrumenting a Linux process through `ptrace` incurred overhead due to the nature of trapping every system call: this requires several context switches between user and kernel space even if Murphy leaves the call unchanged. To get a feel for this overhead, we measured the wall-clock time of running the OpenSSL test suite with and without Murphy instrumentation. First we ran the test suite with no gremlins enabled. This measured the `ptrace` overhead exclusive of any repercussions from manipulating the system calls. This took 34 seconds instead of the non-Murphy baseline of 6 seconds, for a slowdown of about 5.7×. Next we ran the test suite with the non-halting gremlins enabled. This incurred significant overhead: 325 seconds, or a 54× slowdown from the baseline. This was primarily due to *readone* and *writeone*: these gremlins dramatically increased the number of system calls that are actually invoked over the lifetime of a process.

To mitigate the performance impact of adding system calls, we added stateful gremlins similar to *readone* and *writeone*, called *readone_s* and *writeone_s*. These read/write one byte on the first invocation, and also remember the count of bytes that were requested but not read/written. If the next dynamic invocation asks to transmit exactly that remainder, then this suggests that the program under test is noticing the incomplete read/write and looping accordingly. It is likely that the program will continue to do so until its original request is fully satisfied. Therefore, when we see such a compensating invocation, we pass that second call into the kernel unmolested and reset the gremlin state for the next call.

[1] `http://lists.gnu.org/archive/html/bug-bash/2012-01/msg00066.html`, `http://sourceware.org/bugzilla/show_bug.cgi?id=13601`

3.3 Validity

A bug candidate can be a false positive if Murphy simulates behavior which is truly impossible, not merely unusual. This may be due to platform-specific semantics of certain I/O devices or other POSIX mechanisms not reflected in Murphy gremlin logic. For example, Linux pipes are explicitly documented as atomic for small writes (i.e. smaller than `PIPE_BUF`). Therefore the *writeone* gremlin is overly pessimistic for writes to Linux pipes.

Another example is reading from the pseudo-random number source, `/dev/urandom`. Some specifications require that reads from this pseudo-device block until enough system entropy is available to satisfy the entire request. This is not a settled matter, however, and has been debated among highly knowledgeable developers [3]. We suggest that the mere existence of this debate argues in favor of programming defensively, regardless of what the developers may eventually decide.

Beyond validity on any one platform, one goal of Murphy is to identify problematic code before it reaches an environment in which it fails. If code is ported to a new platform, the specialized semantics of one OS may not apply. For example, an OS for embedded devices may have smaller buffers and may make weaker guarantees than our reference platform. Some platforms may not support all of POSIX.1, or may not support the most recently ratified standard. Part of Murphy's value is its ability to identify these potential problems even on a platform where such behavior is impossible. Thus Murphy is especially helpful when cross-platform portability is a goal.

4 Related Work

Our work is closely related to *software fault injection* (SFI), which traps certain calls and introduces faults [6]. Some SFI work actually corrupts memory, registers, or returned data [4]. We return rare-but-legitimate values, never corrupting an otherwise-valid system. SFI often targets only specific areas of the system, using custom device drivers [16] or operating at the boundary between shared libraries and the application [8]. Murphy traps at the `ptrace` level and can intercept all system calls, allowing a much broader range of faults to be injected.

Fuzz testing runs programs on random inputs, often triggering failures due to lax input validation [9]. However, programmers rarely view the kernel itself as potentially disruptive; our work shows the risks of this oversight by "fuzzing" the program from an unexpected direction. In addition, while fuzz inputs are often invalid, Murphy interferes in unusual but technically valid ways.

Dynamic memory-access checkers detect a narrow class of errors relating to pointer abuse [5, 11, 13]. They cannot expose errors that occur only rarely in adverse

environments unless the program is actually run in such an adverse environment. Murphy is complementary, as it creates exactly these adverse environments. Using Murphy and a memory-access checker simultaneously may reveal additional memory bugs that only manifest under the unusual circumstances that Murphy brings forth.

Many testing tools require access to the program's source code. Our approach is purely black box, suitable for robustness testing even of commercial, off-the-shelf (COTS) executables. Another black-box alternative is to add gremlins directly into an operating system, such as by modifying the User Mode Linux (UML) virtual machine [2]. However, this would not provide any more information than we can get via `ptrace`. It could also destabilize components not targeted for testing, making the entire analysis less deterministic.

5 Conclusions and Future Work

Murphy helps application developers trigger, reproduce, and diagnose bugs arising from legitimate but unexpected kernel responses. Our approach uncovers several bugs even in widely deployed and well-tested code. Given this, we anticipate this approach will be even more valuable in the hardening and testing of new software.

Clearly, additional gremlins will expose more classes of bugs. Gremlins that simulate temporary network problems may be especially fruitful. Richer information about system call contexts will allow finer-grained gremlin activation and thus a more targeted hunt for some specific bugs. Additional state tracking by existing gremlins may reduce false positives and improve performance.

Murphy reveals pervasive bugs even in well-tested programs. Nothing we tested handled errnos `EINTR` or `EAGAIN` without failure. This raises an important question: is it naïve for a kernel to return these responses if nothing is going to deal with them correctly? Perhaps instead these types of failures should be squashed in the OS or run-time libraries before returning to the application. Our experience shows that this may be the only practical means to ensure these cases are handled correctly.

We wish to explore the creation of a defensive software-hardening tool that squashes exactly the sort of errors that Murphy simulates. So many programs seem to have problems correctly handling various responses, especially errnos `EAGAIN` and `EINTR`. Therefore perhaps there is a need for such a hardening tool, complete with its own policy language describing how to handle errors (retry, block until success, timeout, no change, etc.). A more comprehensive survey of existing applications' behavior under Murphy would improve our understanding of software's implicit assumptions. This may motivate further research on mitigation strategies.

References

[1] J. Corbet. On vsyscalls and the vDSO. `http://lwn.net/Articles/446528/`, June 2011.

[2] J. Dike. *User Mode Linux*. Prentice Hall, Upper Saddle River, NJ, 2006. ISBN 0131865056.

[3] U. Drepper. short read from `/dev/urandom`. `https://lkml.org/lkml/2005/1/13/485`, Jan. 2005.

[4] S. Han, K. G. Shin, and H. A. Rosenberg. DOCTOR: an integrated software fault injection environment for distributed real-time systems. *Computer Performance and Dependability Symposium, International*, 0:0204, 1995. ISSN 1087-2191. doi:10.1109/IPDS.1995.395831.

[5] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*, pages 125–138, 1991.

[6] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *Computer*, 30:75–82, Apr. 1997. ISSN 0018-9162. doi:10.1109/2.585157.

[7] M. B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *In Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 80–93, 1993.

[8] P. D. Marinescu and G. C. LFI: A practical and general library-level fault injector. In *Proceedings of the Intl. Conference on Dependeable Systems and Networks (DSN)*, Lisbon, Portugal, June 2009.

[9] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. In *In Proceedings of the Workshop of Parallel and Distributed Debugging*, pages ix–xxi. Academic Medicine, 1990.

[10] Z. Miller, T. Tannenbaum, and B. Liblit. Murphy: An environment for advance identification of run-time failures. Technical Report 1770, Department of Computer Sciences, University of Wisconsin–Madison, Apr. 2012.

[11] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *In Third Workshop on Runtime Verification (RV)*, 2003.

[12] P. Nguyen. Loopback devices in linux. `http://csulb.pnguyen.net/loopbackDev.html`, Apr. 2010.

[13] B. Perens. Electric Fence. `http://perens.com/FreeSoftware/ElectricFence/`, Sept. 2010.

[14] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC7)*, Chicago, IL, July 1998.

[15] D. Thain and M. Livny. Parrot: An application environment for data-intensive computing. *Journal of Parallel and Distributed Computing Practices*, 2004.

[16] T. Tsai and R. Iyer. Measuring fault tolerance with the FTAPE fault injection tool. In H. Beilner and F. Bause, editors, *Quantitative Evaluation of Computing and Communication Systems*, volume 977 of *Lecture Notes in Computer Science*, pages 26–40. Springer Berlin / Heidelberg, 1995. ISBN 978-3-540-60300-9. doi:10.1007/BFb0024305.

[17] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 28:2002, 2002.

A Scalable Server for 3D Metaverses

http://sirikata.com

Ewen Cheslack-Postava[1], Tahir Azim[1], Behram F.T. Mistree[1],
Daniel Reiter Horn[1], Jeff Terrace[2], Philip Levis[1], and Michael J. Freedman[2]

[1]Stanford University [2]Princeton University

Abstract

Metaverses are three-dimensional virtual worlds where anyone can add and script new objects. Metaverses today, such as Second Life, are dull, lifeless, and stagnant because users can see and interact with only a tiny region around them, rather than a large and immersive world. Current metaverses impose this distance restriction on visibility and interaction in order to scale to large worlds, as the restriction avoids appreciable shared state in underlying distributed systems.

We present the design and implementation of the Sirikata metaverse server. The Sirikata server scales to support large, complex worlds, even as it allows users to see and interact with the entire world. It achieves both goals simultaneously by leveraging properties of the real world and 3D environments in its core systems, such as a novel distributed data structure for virtual object queries based on visible size. We evaluate core services in isolation as well as part of the entire system, demonstrating that these novel designs do not sacrifice performance. Applications developed by Sirikata users support our claim that removing the distance restriction enables new, compelling applications that are infeasible in today's metaverses.

1. INTRODUCTION

Virtual worlds are three-dimensional graphical environments where people can interact, engage in games, and collaborate. This paper focuses on one particular class of virtual worlds: "metaverses," such as Second Life, where anyone can add new objects to the world by creating 3D models and writing scripts to control them.

Users of today's metaverses miss out on a truly immersive experience because they can see and interact with only the tiny, local region around them. For example, Second Life rendered completely is a fantastic and eye-popping megalopolis, a fictional Shanghai. But users are limited to their local surroundings because Second Life only displays nearby objects and imposes a maximum object interaction distance of ~100 meters. Imposing a distance restriction allows a metaverse to scale easily because a server only shares state and communicates with the servers that manage neighboring regions. But this scalability comes at a cost of user experience. Today's metaverses are lonely and empty [26] and users cannot see what awaits them beyond their interaction range. A distant building in a virtual city should be able to attract a user's attention, and she should be able to select it and send it messages to learn more about it. What is it? Who owns it? Such a simple interaction is impossible in today's metaverses.

In contrast to metaverses, "games," such as World of Warcraft, avoid this problem because the game provider is the centralized author of all world content and the system is application-specific. Central control of content allows the provider to design content around technical limitations. For example, games can precompute scene optimizations (e.g., binary space partition trees for visibility calcuations [31] and imposters for efficient rendering [16]) because game worlds are mostly static. As narrow, tailored applications, games can leverage game mechanics for improved performance: Donnybrook, for instance, showed how focusing updates around the flag bearer in capture the flag leads to lower bandwidth demand [5]. Application-specific optimizations commonly permeate the designs of these systems. For example, World of Warcraft scales to millions of users by dividing users across hundreds of "realms," or copies of the world, and most game content occurs within "instances" where a group interacts with the game privately.

Metaverses differ greatly from games and encounter correspondingly different technical challenges. Instead of static, centrally authored content serving a single application, a metaverse presents a large continous world populated by many user-generated applications whose dynamic behavior, placement, and appearance are controlled by their owners. The tricks used in games are not sufficient in a metaverse: a user could add a skyscraper using a new mesh with unknown behavior, which must quickly be made visible to users and be able to interact with other objects and avatars.

We believe that metaverse users should be able to view and interact with the entire world. However, to support a

world with billions of dynamic user-generated and user-controlled objects, metaverses must somehow limit interaction. Without limits the system would quickly exceed capacity: clients cannot render the entire world in full detail, position updates for every object would saturate a server's outgoing capacity, and even sending a list of all objects to a client would overload a server.

The research contribution of this paper is establishing a novel principle for how to build scalable systems for dynamic, user-extensible virtual environments. Scaling these systems is challenging because the common approach in systems of share-nothing partitionings leads to the very problems we observe in today's metaverses, restricting the view of the world and scope of interaction. Simply removing these limitations leads to unscalable systems. For example, displaying the entire world requires tracking distant objects and therefore requires global queries for objects. Without additional constraints, global queries would require all pairs of servers to exchange queries, limiting the world to just a few servers.

The graphics research community has not tackled this problem in a significant way, typically focusing their systems only on a subset of the requirements of metaverses. For example, render farms are distributed systems that handle large datasets but work offline, and real-time rendering is commonly applied to data that fit in memory on a single host and permit precomputing acceleration data structures. Many ideas from graphics research can be borrowed, but different decisions must be made to work in real-time, in a distributed system, and handle dynamic user-generated content.

This paper presents the design and implementation of a metaverse server that allows clients to see and interact with a large and complex world, even as the system grows to many servers. The key insight for achieving this scalability goal is that a virtual world, being a 3D, geometric environment, has many similarities to the real world. By leveraging real-world-like constraints, the server can simultaneously scale to a large world and provide a natural as well as intuitive experience.

Consider the server's object discovery service, which informs clients about objects in the world. Rather than return the closest objects (which restricts visibility to a small area) or all of the world's billions of objects (which does not scale), the Sirikata server returns objects that have the largest *solid angle*. Solid angle is roughly equivalent to how many pixels an object occupies on screen, so Sirikata returns only objects which could be seen by the user on screen. While this query is global (an avatar can see a building that is many servers away), the number of distant objects and associated network traffic is small. Unlike a distance metric, solid angles neither hide distant, visually important buildings nor do they cause them to suddenly and jarringly pop into existence.

This principle of leveraging real-world constraints underlies all of the server's core systems and is critical to scalability. The server is part of the open-source Sirikata platform [33], already used in applications ranging from virtual museums [3] to social spaces for cancer patients [19]. The paper describes four core components, focusing on how each can scale by applying this principle:

- A server partitioning service that divides a world into regions and maps regions to servers. Assuming an object density distribution similar to the real world leads to a design using a distributed, split-axis kd-tree, with a highly replicated, stable upper tree and distributed lower trees to spread load.
- An object discovery service that prioritizes objects by visual importance, choosing objects with larger solid angles, with a novel distributed data structure and supporting query processor.
- A message routing table that maps object identifiers to servers. Every server can forward to any object in the world, but the routing table scales well because, like the real world, most interactions are local and most objects are stationary.
- A message forwarder that guarantees a minimum throughput even as the number of communicating objects grows very large, decaying smoothly over distance.

We focus on the server partitioning and object discovery services, providing a brief overview of the routing table and forwarder for completeness. We evaluate the first two services in isolation as well as the entire Sirikata server, finding it can scale to large, visually rich worlds with many applications. We also report on our experiences with users developing applications in Sirikata, including some which are difficult or impossible in existing metaverses due to their use of a distance limitation. Finally, while this paper focuses on metaverses, we believe its principles can be applied more generally to systems that bridge physical and virtual environments, such as augmented reality and ubiquitous computing.

2. METAVERSES TODAY: SECOND LIFE

A metaverse is a virtual world that users can extend by adding objects and scripting them to provide interesting applications. This user-extensibility contrasts with games, where all content originates from a central author. A metaverse provides a few basic services with which to build applications:

- storing the location of objects and a reference to the graphical models of objects,
- updating location and model state based on script commands and physics simulation,

- executing object scripts and generating script events, such as timers and inter-object communication,
- telling objects and clients what objects they can see and interact with.

Second Life, today's most popular metaverse, supports over 1 million monthly active users [37]. Its design is representative of how current metaverses decompose internal systems and support applications. Second Life statically partitions land into 256m × 256m regions called "sims," each managed by a different server. A sim's server is responsible for all activity within the sim, including the above services.

Second Life uses distance to control object discovery and inter-object communication. Servers prioritize objects for clients based on multiple factors, but generally clients cannot see objects farther than 100 meters away. Scripted objects (applications) cannot detect objects farther than 15 meters away. The basic communication primitive is a geometric broadcast to all objects within 10, 20, or 100 meters. Second Life also provides a very limited form of unicast which is unsuitable for interactive applications due to its low throughput and high latency.

These restrictions are critical for Second Life to be able scale to its current size. They allow the simulation to be nearly shared nothing: servers only coordinate with their geometric neighbors. Because the region-to-server mapping is static in Second Life, the system cannot respond to variations in load and most sims can support only a small number (40) of avatars. As a result, most of the world lies empty [26].

Relaxing these restrictions may enable virtual worlds to move closer to their imagined potential. Users will be able to view sweeping vistas, and the applications they interact with will be able to span the world. Users will be able to serendipitously discover events throughout the world; players in a game will be able to monitor and aid distant allies; and real-world data will take on new meaning when overlaid onto a virtual replica of Earth. Enabling this vision, however, requires novel distributed systems principles and techniques.

3. SIRIKATA SERVERS

Sirikata servers allow users to see and interact with a large virtual world that runs distributed over many hosts. It breaks the tension between scale and scope by leveraging constraints and restrictions similar to the real world throughout its systems. This section describes the internals of a Sirikata server, as well as its role as one part of a complete metaverse system.

3.1 Sirikata Applications and Overview

Sirikata has a very different application model than Second Life because it allows interaction with distant objects. Sirikata's model greatly resembles the web of today: application objects send code and data to user objects. For example, when an avatar activates a chess board, the board sends the avatar a UI, which exchanges application messages with the board object. This application model makes long-range inter-object communication a key system service. Applications are written in the Emerson language, a JavaScript dialect we developed that introduces mechanisms to address many of the security problems web applications have exposed [25].

Sirikata splits the implementation of a virtual world into three components. Object hosts run the object scripts that implement user-defined applications. Scripted objects populate and move through the world, sending and receiving application-level messages. A content distribution network (CDN) stores and delivers large pieces of data, such as graphical models and textures. Space servers (or just "servers"), the subject of this paper, are responsible for the virtual world itself. They are authoritative on the presence and positions of objects in the world, simulate physics, inform observers of other relevant objects and keep them up-to-date on those objects' positions. Finally, servers are responsible for routing the application-level messages between scripted objects.

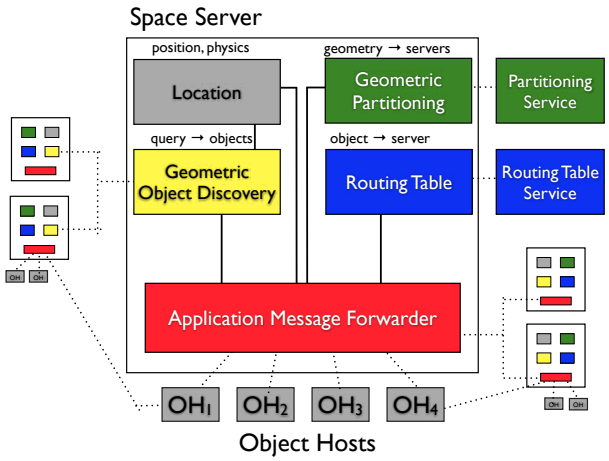
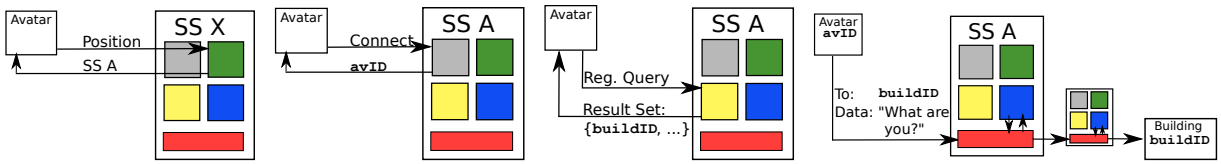


Figure 1: Sirikata server internals

3.2 Server Internals

Figure 1 shows the interaction of object hosts with servers and the internal decomposition of a Sirikata server into its major services. Each server is responsible for a subset of the geometric area of the world. Object hosts connect to the servers which manage the regions occupied by their objects. As shown, a single object host may connect to many servers, and many object hosts may connect to each server. Eliding some ancillary services, such as login/authentication, for brevity, the major services are as follows:



(a) Server discovery (b) Enter world (c) Discover building (d) Message building

Figure 2: Avatar connection and messaging example. Service positions and colors are consistent with Figure 1.

- The **geometric server partitioning** service maintains a distributed data structure for controlling and querying the partitioning of the world across servers (Section 4).
- The **location table** stores the position, orientation, and motion of objects, which the physics engine (based on Bullet [12]) updates. Because this is a traditional component and not novel, this paper does not discuss it in depth.
- The **geometric object discovery** service handles queries to discover other objects, streaming back object identifiers and subscribing the querier for location updates (Section 5).
- The **application message routing** table maintains a durable, consistent, distributed mapping from an object identifier to the space server responsible for it (Section 6.1).
- The **application message forwarder** forwards inter-object (application) messages, using the routing table to determine the next hop. It responds to congestion by giving greater weight to traffic between closer and larger objects (Section 6.2).

Each of these services leverages some property of the physical world in order to scale. For example, the geometric partitioning service leverages the fact that most objects in the world are stationary, such that most changes to its data structure are small and local. As a second example, the message router weights inter-object message flows similarly to the inverse-square falloff of electromagnetic radiation intensity over distance. This weighting provides a natural and intuitive experience: larger and closer objects have greater communication capacity than small and distant ones. Furthermore, the sum of weights for objects in a fixed region converges to a constant, so Sirikata can guarantee that nearby objects cannot be drowned out.

3.3 Example Execution

To explain how Sirikata's services create a working virtual world system, we use the example from Section 1 of a user logging into a world, seeing an interesting building, and selecting it to learn more about it. Figure 2 illustrates what happens.

To log in, the object host running the user's avatar connects to any server and requests an initial position. The space server, SS X, looks up this location with the partitioning service and redirects the avatar to the server managing that region (Figure 2(a)). The avatar connects to and authenticates with the new server, SS A. The server enters the avatar state into the location table and adds entries in the routing table and query processor (Figure 2(b)). To display the world, the avatar registers an object discovery query (Figure 2(c)). The discovery service streams object identifiers and important data, such as position and a URL for each object's graphical model. One such object is the distant, interesting building with object identifier `buildID`.

Figure 2(d) shows how the avatar sends an application-level message to the building object when the user selects it. The message takes three hops. First, the avatar's object host sends it to the server. The server consults its routing table to determine which server is responsible for the building and passes the message to the forwarder. Finally, the destination server forwards it to the building's object host. The building sends a message in reply, containing a web page of information, which an in-world web browser displays.

The next four sections follow the steps of this example and describe the design and implementation of each of the server's core services.

4. SERVER PARTITIONING

As shown in Figure 2(a), when an avatar joins the world, it queries a known server's partitioning service to determine which space server is currently authoritative for its future position. This is necessary because, to support changes in load, Sirikata's mapping of geometric regions to space servers is dynamic.

The geometric server partitioning service manages this mapping with two abstractions. Given a bounding region, it identifies the space servers that manage the region, and given a space server identifier, it returns the bounding region managed by that server. It also chooses when to merge and split regions to balance load across servers, notifying the servers that they should change the region they manage and migrate objects.

The key challenge in implementing the partitioning ser-

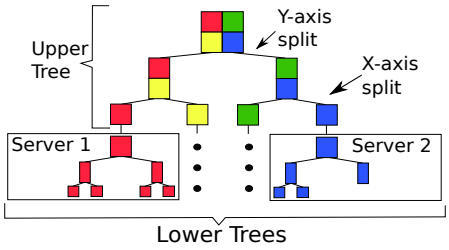


Figure 3: The kd-tree used by the partitioning service and its division into upper and lower trees.

vice is designing a scalable, distributed space-partitioning data structure to efficiently answer bounding-box queries. This data structure should support Earth-scale worlds, make splitting and merging regions for load balancing efficient and simple, and not require major restructuring as objects move, appear, and disappear.

The partitioning service uses a split-axis kd-tree as its basic data structure. Well-known to the graphics community, kd-trees are binary trees that recursively split a region using axis-aligned splitting planes [4]. The partitioning service uses a *split-axis* kd-tree, which always splits regions in equal halves, using the same axis for all splits at a given level of the tree. The split-axis kd-tree cycles between the x- and y- axes to select the splitting planes at successive levels of the tree. Splitting continues as long as the resulting regions (the leaves of the tree) contain more than n objects. Figure 3 shows an example of this process. Split-axis kd-trees are simpler to build and maintain than ones which optimize the placement of split-planes because the plane is not affected by object motion or object distribution. The drawback is that they are deeper than optimally-constructed kd-trees, which reduces their performance for spatial queries.

An experiment quantifies this tradeoff. Using historical and projected real-world population data from 1990–2015 that partitions Earth into ~29 million cells [10], we construct a split-axis kd-tree with at most 40 people per leaf node. The maximum depth of this kd-tree is never more than 50% larger than the optimal kd-tree and the average depth increases by only 20%. As performance evaluations in Section 7.2 later show, this additional tree traversal time is negligible compared to a query's network latency.

4.1 Distributing the Partitioning Service

Although the entire kd-tree of an Earth-sized world can fit in-memory on a single server, the partitioning service must distribute the kd-tree across many hosts for availability, fault-tolerance, and to support high query and update rates. A straw man approach with strongly-consistent replicas of the full kd-tree becomes too costly as regions split and merge for load balancing. The tree has 29 million leaves, and avatar movement can cause frequent splits and merges: large movements of avatars could easily overwhelm a single data structure's ability to process strongly consistent updates.

Depth:	10	12	14	16	18	20
World Pop	6.0	40.0	238.0	966.0	4642.0	22,140.0
Zipf	0.0	0.0	0.0	0.0	1240.4	290,000.0

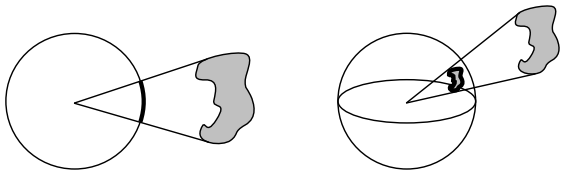
Table 1: Number of changes in the partitioning service's upper tree with different choices of cut depth.

Instead, the partitioning service divides the kd-tree into two parts at a fixed cut depth (Figure 3): an upper tree and a set of lower trees. The upper tree is replicated across all partitioning servers. The lower trees are managed by individual partitioning servers. Each upper tree leaf specifies the domain name of the partitioning server that stores the lower tree rooted at that leaf. Any partitioning service node can answer bounding box queries by traversing the replicated upper tree and forwarding the query to servers managing the relevant lower trees.

This design is effective because the split-axis kd-tree makes the upper tree very stable. Table 1 demonstrates upper tree stability, showing the number of changes in the upper tree for different upper tree depths. Two distributions are tested. The first uses the world population data discussed in this section to count the number of changes in the upper tree as world population increased from 1990–2015. The second is Zipfian, a distribution observed in multiplayer games [6], and uses the same world population cells, assigning each a density of $\frac{D}{i}$ randomly, where $D = 120,000 \frac{objects}{km^2}$ and $i = 1, 2, \ldots$ The table shows changes averaged over ten different Zipf distributions to demonstrate the stability of the upper tree as the densest parts of the world shift around. A sufficiently high cut mostly avoids changes to the upper tree: at a depth of 10 there no changes for the Zipf distribution and only 6 changes over 25 years for the world population data. Even at a depth of 16 there are fewer than 1000 changes — less than 2% of nodes at or above that depth. This low write rate makes it feasible to replicate the upper tree across all partitioning servers.

4.2 Load Balancing

The partitioning service dynamically splits and merges regions to balance server load. It currently uses an ad-hoc strategy: split a region in two when the number of objects inside it exceeds a threshold, and merge two sibling regions if both have fewer objects than $\frac{1}{4}$ the threshold. This approach can create a logarithmic number of lightly-loaded servers if a hotspot occurs within a region that was previously empty. Multiple space server processes can be co-located on the same host to avoid underutilization.



(a) Planar angle (b) Solid angle

Figure 4: Solid angle is the extension of a planar angle to three dimensions. It is defined as the area of an object projected onto a unit sphere centered at the observer.

5. GEOMETRIC OBJECT DISCOVERY

Once connected to the space, an object queries the geometric object discovery service to learn about other objects (Figure 2(c)). Due to memory, network, and display constraints of clients, the server can only return a limited subset of the world. Systems today commonly use distance queries, displaying objects within some distance d, which not only limits what a user can see and interact with, but also leads to jarring discontinuities.

Sirikata relies on two properties of the real world to provide a better set of results than current systems:

Visual perspective: Sirikata uses solid angle queries to prioritize objects that are visually more important. The solid angle of an object is the surface area covered by the object when it is projected onto the unit sphere centered at the observer (Figure 4). It is a measure of how large an object appears to an observer and is roughly equivalent to the fraction of pixels the object would occupy if the display rendered the world in all directions at once. Figure 5 shows the substantial difference between distance and solid angle queries.

Object aggregation: Large collections of small objects, e.g., distant trees, cannot be seen individually, but together contribute significantly to a scene. Sirikata generates simplified aggregate meshes to represent aggregate objects that would otherwise be omitted and the query processor returns these aggregates along with individual objects.

Objects register a query with a simple message specifying the minimum solid angle, i.e., smallest apparent size, of objects that the server includes in the results. The query implicitly includes the querying object's current position. Because solid angle is affected by the querier position and object position and size, query results change due to querier movement, other objects moving or changing size, or objects entering or leaving the world.

Two challenges arise in implementing the solid angle query processor. The first is adapting existing data structures so solid angle queries can be efficiently evaluated on a single host. Section 5.1 presents a novel data structure, the LBVH, that extends Bounding Volume Hierarchies (BVH) by having internal nodes track the largest objects in their subtrees. The second challenge is that solid angle queries are global: they can return an object from anywhere in the world if that object is large enough. Section 5.2 presents a distributed query processor, which addresses the computational and network costs of global queries spanning multiple space servers.

(a) Distance

(b) Solid Angle

Figure 5: Solid angle queries returning the same number of objects as a distance query give a more complete view of the world.

5.1 Efficient Solid Angle Queries

Sirikata efficiently evaluates solid angle queries by extending bounding volume hierarchies, a data structure commonly used in graphics systems for other geometric queries such as distance queries. This section describes this extension, how the same data structure handles standing queries, and how aggregate objects are returned in result queries when individual objects are too small to be returned.

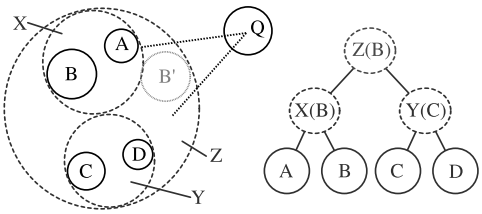


Figure 6: The LBVH tracks the largest leaf of each internal node, allowing it to test query Q against a much smaller virtual object B' instead of Z.

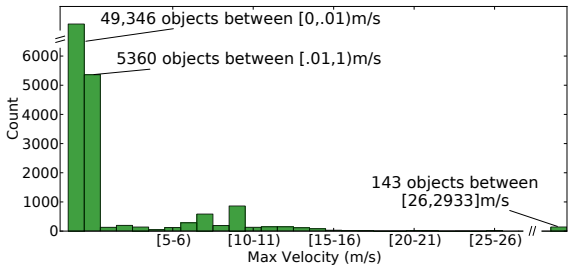


Figure 7: Most objects in a 4 km^2 region of Second Life were not moving. Outlying object speeds are due to objects teleporting.

5.1.1 Solid Angle Queries

Bounding volume hierarchies recursively enclose collections of objects in bounding volumes. This enables efficient query evaluation because entire subtrees are culled if their bounding volume does not satisfy the query [2]. They form a better basis for solid angle query processing than spatial subdivisions (such as binary space partitioning [31] and kd-trees [4]) for three reasons. First, they test each object at most once because each leaf stores an object instead a list of objects overlapping the leaf's region. Second, insertions and removals are efficient regardless of object size. Finally, BVHs support an inexpensive "refit" [23] operation for when objects move: the topology of the tree remains fixed, but the bounding volumes of internal nodes are updated to account for the moving objects.

A bounding sphere hierarchy is a natural starting point because computing the solid angle for spheres is fast and easy. However, bounding sphere hierarchies are inefficient for solid angle queries because they are so conservative: a parent bounding sphere can be much, much larger than its children if those children are distant from one another. In Figure 6, for example, the bounding sphere X is much larger than either A or B. In a simple test using objects collected from Second Life regions, for example, we found evaluating solid angle queries with a BVH visited up to 86% of the tree even when fewer than 20% of leaves were in the result.

Sirikata improves the efficiency of solid angle queries through a new data structure, called the *Largest Bounding Volume Hierarchy* (LBVH). Each LBVH node includes a reference to the largest leaf in the corresponding subtree, as shown in Figure 6. When evaluating whether to traverse a subtree, the query processor tests whether the largest object in that subtree, placed as close to the querier as possible within the bounding volume, would satisfy the query. This test represents the worst case object in the subtree, however, Figure 6 shows that this test is much less conservative (e.g., B' is much smaller than Z). Using an LBVH reduces the cost of a query by 75–90% compared to a BVH.

An LBVH degrades in quality over time as objects grouped in a subtree move apart. To prevent this degradation (and need for possible full LBVH recomputation) the query processor exploits the fact that, just as in the real world, most objects are stationary. A metaverse filled with constantly moving objects is overwhelming, distracting, and difficult to navigate. Data collected from Second Life (Figure 7) show that over 95% of objects in a 4km^2 region do not move over a one hour period. Therefore, the server maintains two LBVHs, one for static and one for dynamic objects. Objects that do not move for one minute are static. The static LBVH rarely changes and so remains efficient. Because so few objects are dynamic, constant factors dominate, and a suboptimal LBVH does not significantly harm performance. The server rebuilds both trees infrequently, currently once an hour.

5.1.2 Standing Queries

The LBVH reduces the cost of evaluating queries one time, but most queries are *standing*: after an initial result, the discovery service sends updates when the result set changes. Simple periodic re-evaluation is wasteful as the same nodes must be re-evaluated even if they have not changed. The query processor avoids this waste by maintaining cuts in the LBVH tree. Each cut stores the LBVH nodes where a query's evaluation terminated. For instance, a cut may store nodes A, B, and Y in Figure 6 if A and B satisfy the query but Y did not. Instead of restarting the query at the root node, the query processor updates the query by traversing its cut nodes. On each update, it tests whether the children of a cut node now appear large enough to satisfy the query or whether a cut node itself now appears too small to satisfy the query, updating the cut and sending notifications to the querier in both cases.

5.1.3 Object Aggregation

Although solid angle queries select better objects, they miss large collections of small objects. For example, Figure 5(b) is actually missing a forest full of trees which the

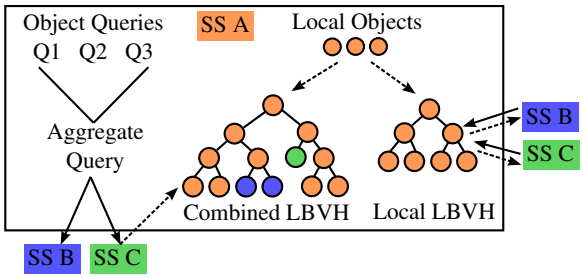


Figure 8: In the distributed query processor, servers query local LBVHs on other servers and incorporate the results into a combined LBVH. The combined LBVH resolves individual object queries.

solid angle query does not capture. To address this, Sirikata performs *aggregation*. The LBVH inherently clusters objects: the server treats each internal LBVH node as an aggregate. Each aggregate is assigned an object identifier and its meshes are combined, simplified, and stored on the CDN. By returning an entire cut instead of only leaves, the querier always has a complete view of the world, although possibly at reduced quality. Aggregates are only visual placeholders; applications cannot communicate with them.

5.2 Distributed Solid Angle Queries

The LBVH evaluates queries locally, but solid angle queries are global. Objects on any server could satisfy a query. Sirikata's distributed query processor addresses the computational and network costs of these distributed global queries. It aggregates queries to reduce inter-server communication and uses a geometric server discovery service to reduce inter-server querying. Figure 8 shows the query processor components of one server.

5.2.1 *Aggregate Queries*

Sirikata exploits the spatial locality of its queriers by conservatively aggregating its local queries into a single outgoing query. The aggregate query's solid angle is the minimum of every query issued to the space server. Instead of a single position, the querier is represented by a bounding sphere of the individual querier positions. To evaluate the aggregate query against an LBVH node, a virtual querier is placed within the bounding sphere as close as possible to the object being tested. Despite being conservative compared to the original queries, having only a single outgoing query reduces computational load on other servers as well as communication cost because it returns only a single set of results.

Each server maintains two LBVHs: local and combined. The local LBVH contains only local objects in the server's region. The combined LBVH includes both local objects and the results of outgoing queries, representing the set of objects in the entire world that might

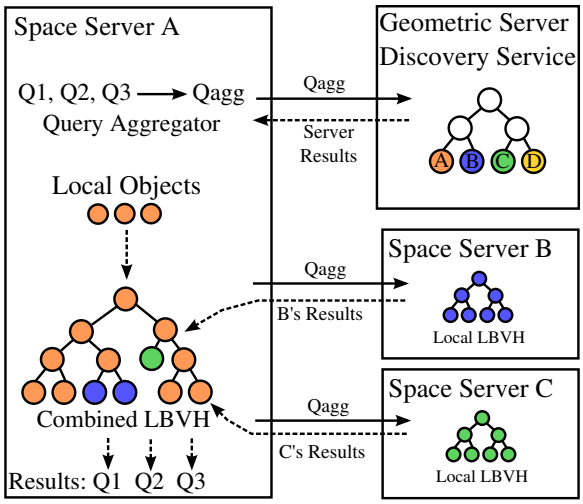


Figure 9: The geometric server discovery service determines which servers must be queried and local LBVHs resolve aggregate queries. The combined LBVH resolves individual object queries.

satisfy queries from objects within the server's region. The server resolves aggregate queries from other servers with the local LBVH and queries from local objects with the combined LBVH.

5.2.2 *Geometric Server Discovery*

Even with aggregate queries, N servers require N^2 server pairs to communicate because queries are global. However, due to distance and object demographics, most servers will have no results for each other. To avoid the waste of N^2 connections, servers discover which other servers to contact through a geometric server discovery service. Figure 9 shows how a server interacts with this service as part of issuing outgoing queries. The geometric server discovery service, like the object discovery service, uses an LBVH. In this LBVH, each leaf points to a space server that can further resolve the query, rather than a single object. Space servers register their aggregate query with the server discovery service and receive a stream of updates specifying which servers to contact whose objects can satisfy their queries.

To evaluate the benefits of server discovery, we simulate a world tiled with objects from Second Life traces. The world grows up to 100km^2 while maintaining constant object density. It is partitioned as described in Section 4 with up to 5000 objects per server and 100 randomly placed queriers per server. Each server's aggregate query is evaluated against every other server's objects to determine if the server must be contacted. Figure 10 shows the results for several query angles, specified in pixels occupied on a 1024x1024 screen with 60° field of view. The number of servers contacted grows more slowly than the total number of servers, eventually

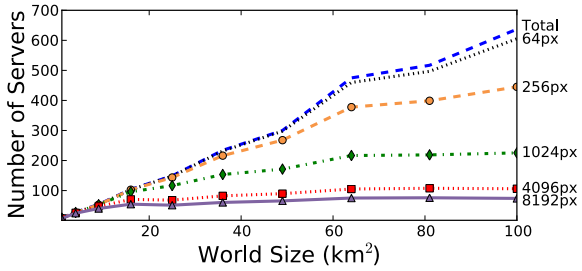


Figure 10: Average number of servers contacted with aggregate queries. Common query angles (1024 to 4096 pixels) require far fewer than the maximum.

reaching a maximum value since the world is growing in size, but not density. Currently, the server discovery service runs on a single host because a single host can easily support thousands of space servers and regenerating its state in the case of failure takes at most a few seconds.

5.3 Evaluation

Figure 11 evaluates the visual effect of using solid angle queries, with a small town scene with 10,000 objects — houses, streets, terrain, and trees. The figure shows images with ~3,000 objects for distance queries and Sirikata's queries as well as the full image (all 10,000). Distance queries miss important objects like the terrain. As Figure 11(b) shows, solid angles with aggregates complete the picture, including detail — such as the distant forest — that is lost otherwise, allowing a client to see a compelling approximation of the entire world.

6. ROUTING AND FORWARDING

Returning to Figure 2(d), after an avatar discovers the building, it interacts with it through application messages. Space servers are responsible for delivering these messages to the proper object host. This delivery has two steps: routing, to determine which space server covers the region the destination is in, and forwarding, delivering the message to that server. Section 6.1 describes routing; Section 6.2 describes forwarding.

6.1 Application Message Routing

Two properties shape the application message routing table's design. First, any pair of objects should be able to communicate, even distant ones. Unlike games, which can be carefully engineered to use only local messaging, metaverse applications may need to communicate over long distances. Second, objects may move freely about the world. Long-range object communication requires that the routing table on any space server be able to answer a query for any object in the world. Object mobility suggests the use of a flat object identifier namespace.

(a) Distance (3,000 objects)

(b) Sirikata (3,000 objects)

(c) Ideal (10,000 objects)

Figure 11: Comparison of Sirikata (solid angles w/aggregates), distance-based queries, and the ideal scene. Sirikata's combination of solid angle queries and aggregation allows it to display the complete world with a fraction of the objects.

A simple key-value lookup, mapping from object identifier to destination server, satisfies these requirements. Sirikata stores routing records in a separate scalable, key-value store [29]. This approach benefits from simplicity, reusing existing software to maintain a single, globally consistent routing record for each object.

To mitigate the cost of a round trip to the backend store for each message, Sirikata heavily caches routing records on each space server, relying on three factors. First, each record is small, only 34 bytes, containing the object's identifier, the space server it is connected to, and metadata the forwarder uses to weight flows. Millions of records can be cached in only tens of megabytes. Second, object mobilitiy is limited. Like in the real world, few objects move in the virtual world (Figure 7). Correspondingly, their routing records do not change. Finally, by caching records when objects migrate, a server can cheaply forward messages it receives due to stale routing entries. It also sends a control message back to the source space server with the new routing record.

6.2 Application Message Forwarding

Once a server has the next hop for a message in its routing table, the forwarder is responsible for deciding when to send it. Under low load, forwarder behavior is simple — forward all packets — but as demand exceeds capacity, it must choose the order in which to forward packets and which packets to drop. Unlike games, where there is only one application that can be designed around network limitations, Sirikata must support many concurrent applications with unknown traffic patterns.

Sirikata draws inspiration from the real world by weighting inter-object flows using an equation similar to the falloff of electromagnetic radiation. This gives closer and larger object pairs a greater portion of bandwidth even under heavy congestion. The challenge of this approach is that it requires distributed state spread across many servers (objects' positions and sizes) and traditional techniques for enforcing these weights are too expensive as the number of communicating object pairs grows.

As discussed and evaluated by Reiter-Horn [18], Sirikata uses heuristics to approximate object positions and sizes and leverages ideas from Core Stateless Fair Queueing to enforce weights it assigns between flows. Reiter-Horn demonstrates that four properties of the Sirikata forwarder provide a much more natural, intuitive experience for users. First, weights are always non-zero, so objects are always able to communicate. Second, just as two people step closer to hear more clearly, objects can simply move closer for higher throughput because weights fall off gradually with distance. Correspondingly, throughput drops as objects separate, but never drops suddenly. Third, through careful selection of the weight function, a minimum quality of service is guaranteed between a pair of fixed objects, even if all other object pairs are trying to communicate at full capacity. A close pair of objects cannot be drowned out. Finally, the forwarder ensures high utilization: if there is only one flow, it can use the full server capacity.

	Latency
Space Server to Upper Tree Latency	1498 μs
Upper Tree Lookup	24 μs
Upper Tree to Lower Tree Latency	848 μs
Lower Tree Lookup	137 μs

Table 2: Latency breakdown for partitioning service queries with a cut depth of 20 on a LAN.

7. PERFORMANCE EVALUATION

Previous sections motivated and evaluated the design decisions in each of the server's services. This section evaluates the performance of their implementations, as well as the end-to-end performance of the entire server. Evaluations were performed on a cluster with 2.4GHz Xeon E5620 CPUs, 8GB RAM, and 1Gbps NICs.

7.1 Implementation

The core Sirikata system is currently ~102,000 lines of code. The space server and services described in this paper are ~22,000 lines of code. The object host, including the graphical client and scripting libraries, are ~45,000 lines of code. The remaining ~35,000 lines of code are in shared utilities. To leverage multicore processors, the space server is highly multithreaded: the current implementation has eight active threads.

7.2 Partitioning Service Performance

The key metric for the partitioning service is latency: how quickly can the system determine the servers responsible for a point or region? Table 2 shows the breakdown of latency for a query using the world population data described in Section 4 and a cap of 40 objects per server. The table separates the two network hops because the server used to query the upper tree is unlikely to contain the lower tree covering the queried region. The latency is ~2.5 ms, of which ~2.3 ms is two RTTs of network latency.

7.3 Geometric Query Performance

Figure 12 shows the query update rate of Sirikata's geometric object query processor, in queries per second, for a world generated from measured Second Life data. The scene contains 100,000 static objects and 1,000 queriers, generated by tiling Second Life data at a higher density to cover $\frac{1}{4}$km^2. All queriers use the same solid angle parameter and follow measured avatar paths.

Cuts reuse query state from the previous iteration and

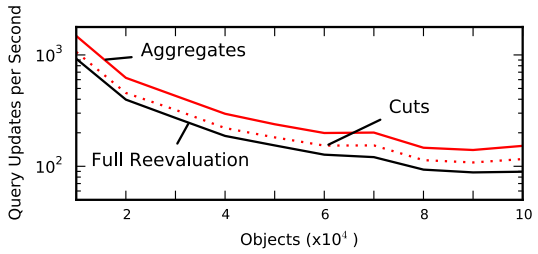


Figure 12: Query updates per second, as the number of objects increases. Cuts improve query update rate by 20% over full reevaluation and cuts with aggregates improve the rate by 56%.

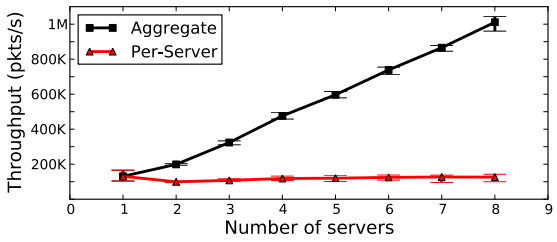


Figure 13: Egress capacity of the space as the number of servers increases, using a uniform workload. Values are the median of a 5-minute period, with 99th percentile error bars.

	Latency	Max Rate	Throughput
Local	324 μs	144,632 pps	384.18 Mbps
Remote	672 μs	71,074 pps	320.25 Mbps
Remote+Lookup	974 μs	38,775 pps	165.79 Mbps

Table 3: Space server forwarding performance. The system can enforce fairness at fine granularity without sacrificing performance.

substantially improve performance over periodically reevaluating full queries, updating about 20% more queries per second. Aggregates improve performance further, with 56% higher query throughput than full reevaluation. Aggregates avoid many solid angle tests at leaves: once one leaf is included in the cut, all its siblings must also be included.

7.4 Routing and Forwarding Performance

We evaluate router and forwarder latency, forwarding rate, and throughput with microbenchmarks between pairs of objects. Latency measures the application-level ping time for 64-byte messages with idle servers. Forwarding rate measures how fast a server can forward 64-byte messages. Throughput measures the maximum inter-object throughput using 1KB messages.

We measure three forwarding paths: local, remote, and remote+lookup. The local path handles messages between objects connected to the same server. The remote path handles messages between objects connected to different servers where the destination routing entry has been cached, passing through inter-server queues and requiring an additional network hop. The remote+lookup path is like remote but also requires a routing table lookup.

Table 3 shows the results. For local messages, the space server can process about 145,000 messages per second and has a latency of about 325μs. A routing service lookup triples message latency, cuts the forwarding rate by 75%, and reduces throughput by 57%. This demonstrates the need for a routing table cache. These demonstrate that the system can enforce fairness while maintaining high performance.

7.5 Scaling Performance

We examine how server performance scales when exercising all of the services under a simple, general workload. This workload consists of objects with a random size, position, and solid-angle query parameter. 95% of the objects are stationary and the rest periodically teleport to a random position and set a random velocity. All objects send 150 30-byte messages per second to a random object they can see. Each server covers a 10km by 2km by 4km regions and servers are added linearly. Object density remains uniform: more objects are added with each additional server.

Figure 13 shows the results for up to space servers with many more objects hosts generating traffic. As the size of the world increases, the aggregate egress capacity of the world increases roughly linearly. While objects in this workload can and do interact with distant objects, e.g., interesting buildings, most of their interactions are with nearby objects, as one would expect. The small drop from one to two servers is due to routing some traffic across server boundaries. While this workload may not exactly model a real-world scenario, it shows that the server can scale up to larger spaces without limiting either a client's view of the world or its ability to communicate with distant objects.

8. APPLICATION EVALUATION

Performance and scalability are only useful if the ability to see and interact with the entire world leads to engaging applications that are not possible in today's systems. We hired 11 undergraduate computer science students for the Summer of 2011 to develop applications in Sirikata. In a few weeks, they created a variety of applications and systems, including:

Minimap - a top-down display of objects and events,

Wiki-world - embedded Wikipedia articles from object metadata,

Escrow - secure object and virtual currency exchange,

Games, including a Pokemon-based RPG, a 3D Pacman clone, 3D tower defense, and a Minecraft clone,

Procedural generators for both city road networks and building layouts, and

Phys-lib, a library for customizable physics with an example billiards application.

These applications demonstrate that Sirikata supports a variety of metaverse applications running concurrently, including some that would be very difficult or impossible in current systems. We focus on two examples, Minimap and Wiki-world.

8.1 Minimap

Minimap presents a 2D top-down view of objects and events as thumbnails. Geometric queries seed the map and it uses messaging with other objects to register events and descriptions. Events may include code to be executed by the avatar, for example to register it as a player in a game. Users can also specify regions to aggregate when zoomed out and share them with others, for example assigning a neighborhood a name. Solid angle queries and long-distance messaging make Minimap feasible, allowing it to give a view of the world beyond its immediate surroundings. In contrast, Second Life requires a separate, centralized 2D map service, which users cannot extend or improve.

8.2 Wiki-world

Wiki-world is similar to the application in Section 1: users can click on objects to learn about them through embedded Wikipedia articles. Executing within an avatar's script, Wiki-world collects search terms by messaging the object and from user-specified tags on the CDN, and presents search results in a 2D interface. The Sirikata server enables Wiki-world for the entire world by allowing discovery of, and communication with, large, distant objects such as the building in Figure 2.

9. RELATED WORK

Sirikata builds on a combination of ideas from distributed systems, networking, and graphics. This section compares Sirikata to existing metaverse systems and reviews prior work that informs the design of its services.

9.1 Metaverses

Metaverses differ from most application-specific virtual worlds, such as those for games and visualization, because they present a single, contiguous world comprised of user-generated content. Scalability tricks used in those systems do not translate directly to metaverses. Most metaverses, including ActiveWorlds [1], Second Life [30] (as well as its open source counterpart OpenSim [27]), and Blue Mars [7], have made the same compromises to enable scalability, choosing to limit visibility and interaction to a small distance in order to scale. The distributed scene graph [22] improves the scalability of OpenSim by running system services across multiple hosts. However, the limits on visibility and interaction are still present.

9.2 Partitioning Service

A few early virtual world games, such as Asheron's Call, dynamically partitioned the world across a cluster of servers [28]. However, most systems use sharding or precomputation to achieve scalability. World of Warcraft divides users into hundreds of "realms," or copies of the world [39]. RING models the world as a large piece of global state, but prunes object updates based on precomputed visibility within a static environment [15]. These techniques do not apply to metaverses where there is a single, contiguous world with dynamic, user-generated content.

Liu et al. make design choices similar to Sirikata in applying binary space partitioning for managing load [24], but no global data structure is maintained. Chen et al. use fixed partitioning, but exploit and encourage local communication by clustering neighboring regions onto hosts [9].

9.3 Discovery Service

Practically all systems today limit object discovery by distance, sometimes referred to as range queries or area of interest (AOI) [6, 20]. Other systems use application-level information [5], disjoint region splitting [21], perceptual limitations [32], or visibility [36] to reduce load. Chaudhuri et al. describe a system for rendering large worlds, guaranteeing a fixed bandwidth cost, given a maximum velocity [8]. All these approaches assume either local interaction or a static environment. Sirikata's discovery service builds on a long history of spatial data structures. It extends the the well-understood bounding volume hierarchy [11] and applies it in a distributed setting to perform efficient, global, solid angle queries.

9.4 Routing and Forwarding

Routing and forwarding is a minor concern in most current metaverse systems because local discovery limits communication to local objects. Sirikata enables communication between any pair of objects. Although most services in Sirikata leverage geometry, routing uses a flat namespace of object identifiers to break any dependency on the assumption of stationary objects. Existing techniques [38], such as distributed hash tables [34, 13], are used to scale identifier lookups. Since object changes often have many observers, many systems use multicast to reduce network traffic [14, 17, 15]. Sirikata focuses on unicast application-level traffic, although

multicast could be applied for sending location table updates. Sirikata's forwarder builds on core stateless fair queueing (CSFQ) [35] to manage weighted flow fairness with minimal state [18].

10. CONCLUSION

This paper argues that metaverses have failed to meet their imagined potential: rather than large, vibrant social spaces, they are quiet, lifeless, and desolate. In order to scale to support a large world, existing systems prevent users from seeing that world, constraining them to a small, local area.

The Sirikata metaverse server allows users to see and interact with a large and dynamic virtual world. Properties of the real world inform its design and provide constraints that enable scalability with intuitive implications: observers have limited resolution, objects are mostly static, and object density is in a small, fixed range. Sirikata enables applications that are very difficult to build in existing systems without sacrificing performance.

The Sirikata metaverse server is one component of a larger vision. Many challenges arise in the object host and CDN described in Section 3, from load balancing objects in a distributed object host to supporting progressive loading of content from the CDN. We believe distributed graphical systems are an underexplored area of research and significant work remains at the intersection of the graphics and systems communities. As computing increasingly moves into the physical world, in the form of augmented reality, cyber-physical systems, and ubiquitous computing, we believe the principles Sirikata has begun to explore may inform the design and implementation of these increasingly important classes of systems.

Acknowledgments

Thanks to our undergraduate developers and testers — Kotaro Hoshino, Jiwon Kim, Elliot Conte, Angela Yeung, Kevin Chung, Emily Ye, Alex Quach, Ben Christel, Will Monroe, Zhihong Xu, and Gabrielle Chen — for spending their summer building applications in Sirikata. Thanks also to Xiaozhou Li, Bhupesh Chandra, Lily Guo, and Patrick Horn for their valuable contributions to Sirikata. We would also like to thank the anonymous reviewers and our shepherd, John Regehr, for their helpful comments.

This work is funded by the National Science Foundation (NeTS-ANET Grants #0831163 and #0831374) and conducted in conjunction with the Intel Science and Technology Center – Visual Computing.

11. REFERENCES

[1] `http://www.activeworlds.com/`.

[2] J. Arvo and D. Kirk. A survey of ray tracing acceleration techniques. *An Introduction to Ray Tracing*, 1989.

[3] H. Bennetsen. Sirikata: Old stuff in new ways. Lecture at Media X at Stanford University, November 2009.

[4] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9), 1975.

[5] A. Bharambe, J. R. Douceur, J. R. Lorch, T. Moscibroda, J. Pang, S. Seshan, and X. Zhuang. Donnybrook: Enabling large-scale, high-speed, peer-to-peer games. In *SIGCOMM Comput. Commun. Rev.*, Aug. 2008.

[6] A. Bharambe, J. Pang, and S. Seshan. Colyseus: a distributed architecture for online multiplayer games. In *Networked Systems Design and Implementation*, May 2006.

[7] `http://www.bluemars.com/`.

[8] S. Chaudhuri, D. Horn, P. Hanrahan, , and V. Koltun. Image-based exploration of massive online environments. Technical Report CSTR 2009-02, Stanford, 2009.

[9] J. Chen, B. Wu, M. Delap, B. Knutsson, H. Lu, and C. Amza. Locality aware dynamic load management for massively multiplayer games. In *Principles and Practice of Parallel Programming*. ACM, 2005.

[10] Gridded population of the world, version 3. `http://sedac.ciesin.columbia.edu/gpw`. Socioeconomic Data and Applications Center (SEDAC), Columbia University.

[11] J. H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 1976.

[12] E. Coumans. Bullet physics engine. `http://www.bulletphysics.org`.

[13] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a DHT for low latency and high throughput. In *Networked Systems Design and Implementation*, 2004.

[14] E. Frécon and M. Stenius. Dive: a scaleable network architecture for distributed virtual environments. *Distributed Systems Engineering*, 1998.

[15] T. A. Funkhouser. RING: A client-server system for multi-user virtual environments. In *Symp. on Interactive 3D Graphics*, Apr. 1995.

[16] T. A. Funkhouser and C. H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. ACM SIGGRAPH, 1993.

[17] L. Gautier and C. Diot. Design and evaluation of MiMaze, a multi-player game on the internet. *ICMCS*, June-July 1998.

[18] D. R. Horn. *Using a Physical Metaphor to Scale Up Communication In Virtual Worlds*. PhD thesis, Stanford University, 2011.

[19] M. Hoybe. BE community: Bridging social isolation for teenagers & young adults with cancer. Games for Health, 2011.

[20] S.-Y. Hu, J.-F. Chen, and T.-H. Chen. Von: A scalable peer-to-peer network for virtual environments. *Network, IEEE*, 2006.

[21] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *INFOCOM*, Mar. 2004.

[22] D. Lake, M. Bowman, and H. Liu. Distributed scene graph to enable thousands of interacting users in a virtual environment. NetGames, 2010.

[23] C. Lauterbach, S. Yoon, and D. Manocha. RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs. In *IEEE Symp. Interactive Ray Tracing*, Sept. 2006.

[24] H. Liu and M. Bowman. Scale virtual worlds through dynamic load balancing. In *Distributed Simulation and Real Time Applications*, 2010.

[25] B. F. T. Mistree, B. Chandra, E. Cheslack-Postava, P. Levis, and D. Gay. Emerson: Accessible scripting for applications in an extensible virtual world. In *OOPSLA*, 2011.

[26] `http://nwn.blogs.com/nwn/2010/01/empty-beauty-of-second-life.html`.

[27] `http://www.opensimulator.org/`.

[28] J. Quimby. Massively multiplayer gameplay system

implementation. Game Developers Conference, 2002.
[29] http://redis.io/.
[30] http://www.secondlife.com/.
[31] R. Shumacker, R. Brand, M. Gilliand, and W. Sharp. Study for applying computer-generated images to visual simulation. Technical Report AFHRL-TR-69-14, U.S. Air Force Human Resources Lab, 1969.
[32] S. K. Singhal. *Effective remote modeling in large-scale distributed simulation and visualization environments*. PhD thesis, Stanford University, 1997.
[33] http://www.sirikata.com/.
[34] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. 2001.
[35] I. Stoica, S. Shenker, and H. Zhang. Core-stateless fair queueing: achieving approximately fair bandwidth allocations in high speed networks. 28(4), 1998.
[36] S. J. Teller and C. H. Séquin. Visibility preprocessing for interactive walkthroughs. In *ACM SIGGRAPH*, 1991.
[37] The Second Life Economy in Q2 2011. http://community.secondlife.com/t5/Featured-News/The-Second-Life-Economy-in-Q2-2011/ba-p/1035321.
[38] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes no longer considered harmful. In *Operating Systems Design and Implementation*, 2004.
[39] http://www.wowwiki.com/Realm, 2011.

Granola: Low-Overhead Distributed Transaction Coordination

James Cowling
MIT CSAIL

Barbara Liskov
MIT CSAIL

Abstract

This paper presents Granola, a transaction coordination infrastructure for building reliable distributed storage applications. Granola provides a strong consistency model, while significantly reducing transaction coordination overhead. We introduce specific support for a new type of *independent* distributed transaction, which we can serialize with no locking overhead and no aborts due to write conflicts. Granola uses a novel timestamp-based coordination mechanism to order distributed transactions, offering lower latency and higher throughput than previous systems that offer strong consistency.

Our experiments show that Granola has low overhead, is scalable and has high throughput. We implemented the TPC-C benchmark on Granola, and achieved 3× the throughput of a platform using a locking approach.

1 Introduction

Online storage systems run at very large scale and typically partition their state among many nodes to provide fast access and sufficient storage space. These systems need to provide persistence, availability, and good performance.

It is also highly desirable to run operations as *atomic transactions*, since this greatly simplifies the reasoning that application developers must do. Transactions allow users to ignore concurrency, since all operations appear to run sequentially in some serial order. Most distributed storage systems do not provide serializable transactions, however, because of concerns about performance and partition tolerance. Instead, they provide weaker semantics, e.g., eventual consistency [14] or causality [26].

This paper presents Granola, an infrastructure for building distributed storage applications where data resides at multiple storage repositories. Granola supports atomic transactions, and provides serializability across all operations. Granola also provides persistence and high availability, along with low per-transaction overhead.

Granola provides transaction ordering, atomicity and reliability on behalf of storage applications that run on the platform. Applications specify their own operations, and Granola does not interpret operation semantics. Granola can thus be used to support a wide variety of storage systems, such as databases and object stores. Granola implements atomic *one-round* transactions. These execute in one round of communication between a user and the storage system, and are used extensively in online transaction processing workloads to avoid the cost of user stalls [7,20,32].

Granola supports three classes of one-round transactions. *Single-repository transactions* execute on a single storage node; we expect that most transactions will be in this class, since data is likely to be well-partitioned. *Coordinated distributed transactions* execute atomically across multiple storage nodes, and commit only if all participants vote to commit; these transactions are what is provided by traditional two-phase commit. We also support a new transaction class, which we term *independent distributed transactions*. These execute atomically across a set of nodes, but do not require agreement, since each participant will independently come to the same commit decision. Examples include an operation to give everyone a raise, an atomic update of a replicated table, or a read-only query that obtains a snapshot of distributed tables.

Granola uses a timestamp-based coordination mechanism to provide serializability for single-repository and independent transactions *without* locking, using clients to propagate timestamp ordering constraints between repositories. This provides a substantial reduction in overhead from locking, log management and aborts, and a consequent improvement in throughput. Granola provides this lock-free coordination protocol while handling single-repository and independent transactions, and adopts a lock-based protocol when handling coordinated transactions. Granola's throughput is similar to existing state-of-the-art approaches when operating under the locking protocol, but significantly higher when it is not.

Granola provides low latency for all transaction types:

we run single-repository transactions with two one-way message delays plus a stable log write, and both types of distributed transactions usually run with only three messages delays plus a stable log write. This is a significant improvement on traditional two-phase commit mechanisms, which require at least two stable log writes, and improves even on modern systems such as Sinfonia [7], which requires at least four one-way message delays (for a remote client) and one stable log write.

Our experiments show that we can provide 3× greater throughput on the TPC-C benchmark compared to using a locking approach.

2 Transaction Model

Granola supports *one-round transactions*, which are expressed in a single round of communication between the client and a set of repositories; we refer to the set of repositories as transaction *participants*. The client application specifies what operations to execute and Granola ensures that these are executed atomically at all participants.

One-round transactions are distinct from general database transactions in two key ways: One-round transactions do not allow for interaction with the client, where the client executes multiple sub-statements (e.g., queries) before issuing a transaction commit; transactions are instead specified as a single operation that executes atomically at each participant. One round transactions also execute to completion at each participant, with no communication with other repositories, apart from at most a single commit/abort vote. Despite these restrictions, one-round transactions are still a powerful primitive. They are used extensively in online transaction processing workloads to avoid the cost of user stalls [7, 20, 32], and map closely to the use of stored procedures in a relational DBMS.

2.1 Why Independent Transactions?

As mentioned, Granola supports three types of one-round transactions: single-repository, coordinated, and independent transactions; this last category is a primary contribution of the Granola protocol.

Coordinated transactions incur significant cost for locking and undo-logging; previous studies have estimated overhead to be 30–40% of CPU load under typical workloads [18]. These also incur overhead when retrying transactions that block or abort due to contention [7]. Our motivation for independent transactions was to explore the most powerful primitive we could provide without incurring this overhead. Independent transactions execute atomically across a set of participants, but do not require locking or undo-logging, and do not contend with other transactions. This provides us a significant performance advantage, as seen in Section 6.2.

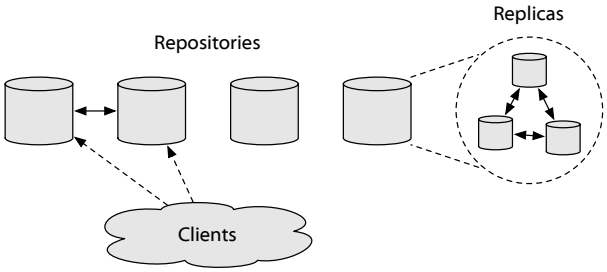


Figure 1: System topology.

Our approach was influenced by the H-Store project [32], which identified a large class of operations in real-world applications, deemed as *one-shot* and *strongly two-phase*, that fit our independent transaction model. This work did not provide a functional protocol for coordinating independent transactions in a distributed setting [19], however. To our knowledge, no previous system provides explicit support for independent transactions.

Independent transactions are appropriate for distributed operations where all participants will make the same local decision whether to commit or abort. This includes distributed read-only transactions, common in read-heavy workloads when data spans multiple partitions, or transactions where the commit decision is a deterministic function of shared data. Application developers commonly replicate tables when partitioning data, to ensure that the majority of transactions are issued to a single partition [20, 32]; independent transactions can be used to atomically update replicated data, or to execute distributed transactions predicated on replicated data.

The common TPC-C benchmark [6], designed to be representative of a typical online transaction processing workload, can be partitioned so that operations consist entirely of single-repository transactions and independent transactions [32]. For example, `new_order` transactions only abort if a request contains an invalid item number, which can be computed locally if the Item table is replicated at each participant. Modifications to the Item table could also be performed using independent transactions.

3 Architecture and Assumptions

This section describes our architecture and application interface. It also discusses our assumptions.

3.1 Architecture

Granola contains two types of nodes: *clients* and *repositories*. Repositories are the server machines that store data and execute transactions, while clients interact with the repositories to issue transaction requests. Repositories communicate with one another to coordinate transactions,

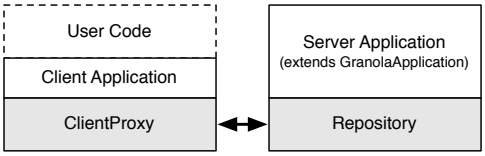


Figure 2: Logical structure.

whereas clients typically interact only with the repositories. This topology is illustrated in Figure 1. Each repository is comprised of a number of replica nodes, to provide reliability; we discuss replication in Section 5.1.

Applications link against the Granola library, which provides functionality for correctly ordering transactions and delivering them reliably to each server application. Applications are layered atop the Granola client and repository code, as shown in Figure 2.

3.2 Client Organization

The *client application* code is provided by the application developer, and supports the desired interface to user code, e.g., support for queries in a database system. The client application is responsible for determining which repositories are required for a given user request, and choosing which transaction class to use.

The application issues transactions by interacting with the Granola *client proxy*, using the interface shown in Figure 3.[1] The application specifies the repository ID (RID) for each participant, the operation to run at each participant, and whether the transaction is read-only. The client proxy interacts with the Granola repository code at the participants, and handles all other client-side functionality, including providing a *TID* (transaction identifier) for each request. The TID uniquely identifies each request, and consists of the client ID and a sequence number that increases for each request from that client. The client proxy also manages timestamps, which are used to ensure serializability as discussed in Section 4.

3.3 Server Organization

The Granola repository code prepares and executes transactions by making upcalls to the server application. The server application must implement the `GranolaApplication` interface shown in Figure 4. Application upcalls supply the transaction operation and retrieve a result, both of which are uninterpreted by Granola. The server application runs in isolation at each repository, and does not need to communicate directly with other repositories, since this functionality is provided by the

[1]While we show a blocking interface for the client proxy, the user can issue multiple requests concurrently; Granola determines the relative ordering of concurrent transactions.

```
// issue trans to given repository
// writes result into provided buffer
void invokeSingle(int rid, ByteBuffer op,
        boolean ro, ByteBuffer result);

// issue trans to set of repositories
void invokeIndep(List<Integer> rids,
        List<ByteBuffer> ops, boolean ro,
        List<ByteBuffer> results);

// issue trans to set of repositories
// returns application commit/abort status
boolean invokeCoord(List<Integer> rids,
        List<ByteBuffer> ops,
        List<ByteBuffer> results);
```

Figure 3: Client API.

Granola repository code. We discuss the use of the server API in Section 4, and the recovery interface in Section 5.4.

3.4 Assumptions

For the purposes of this paper, we assume that the set of repositories is fixed and well-known; system reconfiguration is outlined in our extended description of the protocol [13], along with other protocol details.

While each repository can process many transactions in parallel, each application upcall is executed sequentially, saving considerable overhead over the cost of latching and concurrency control [32]. This approach works well assuming in-memory workloads, as is common in most large-scale transaction processing applications [32]. Multiple repositories may be colocated on a single machine to take advantage of multicore systems.

Granola tolerates crash failures. Our replication protocol depends on replicas having loosely synchronized clock *rates* [25]. We depend on repositories having loosely synchronized clocks for performance, but not correctness.

4 Protocol Design

This section describes the Granola protocol. We first discuss the timestamps used to provide serializability, followed by our protocols for the three transaction classes.

Each repository runs in one of two modes. When there are no coordinated transactions running at a repository it runs in *timestamp mode*. Sections 4.3 and 4.4 describe our protocols for single-repository and distributed transactions as they work in timestamp mode. When the repository receives a request for a coordinated transaction it switches to *locking mode*. Section 4.5 describes how the system runs in this mode and how it transitions between modes.

The repository interacts with the server application differently in the two different transaction modes. In time-

```
Independent Interface
// executes transaction to completion.
// returns false if lock conflict
boolean run(ByteBuffer op, ByteBuffer result);

Coordinated Interface
// runs to commit point and acquires locks.
// returns COMMIT/ABORT vote or CONFLICT
// result is empty unless returning ABORT
AbortType prepare(ByteBuffer op, long tid,
        ByteBuffer result);

// commits trans and releases locks
void commit(long tid, ByteBuffer result);

// aborts trans and releases locks
void abort(long tid);

Recovery Interface
// force-acquires any locks that could be
// required at any point in the serial order
// returns true if no conflict
boolean forcePrepare(ByteBuffer request,
        long tid);
```

Figure 4: Server API.

stamp mode, all transactions are executed using the `run` upcall, which executes the transaction to completion. The `prepare` upcall is used for distributed transactions when in locking mode, to acquire locks on the transaction and determine the commit or abort vote. The response to the `prepare` upcall can indicate COMMIT, ABORT, or CONFLICT. COMMIT indicates that the application has acquired the locks needed by the request while CONFLICT means that some locks cannot be acquired and therefore the client should retry the transaction. ABORT means that the application has decided to abort the transaction based on application logic, e.g., the application refuses to decrement an account balance because the balance is too small. If the application returns ABORT it can also include additional information for the client in the `result` buffer. The ABORT response occurs only for coordinated transaction requests.

In the following description we note where a *stable log write* is required. This step involves the primary replica executing state machine replication, as described in Section 5.1. Unless specified, no other replication occurs and communication is solely between the primary replicas at each repository.

4.1 Timestamps

Granola uses timestamps to order distributed transactions without locking. Each transaction is assigned a timestamp, which defines its position in the global serial order. A transaction is ordered before any transaction with a larger timestamp; if two transactions have the same timestamp, the transaction with the lower TID is ordered first.

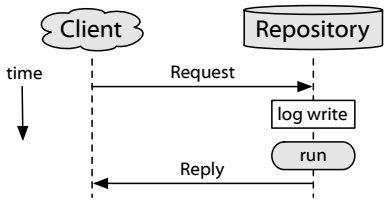


Figure 5: Timeline for single-repository transactions.

Repositories select the timestamp for each transaction based on their clock. Repositories exchange timestamps before committing a given transaction, to ensure that they all assign it the same timestamp. Each transaction result sent to the client contains the timestamp for that transaction, and each request from the client contains the latest timestamp observed by the client, ensuring that timestamp dependencies are maintained. We explain how timestamps are used in the following sections.

4.2 Client Protocol

The client proxy receives transaction invocation requests from the client application via the interface specified in Figure 3. Each client proxy maintains *highTS*, the highest timestamp it has observed in a transaction response, initially 0. The client proxy issues a transaction REQUEST to the repositories specified by the client application, along with the highTS value and the TID.

The client proxy then waits for REPLY messages from the participants. If it receives COMMITs from all participants, it returns the results to the client application. If the proxy receives an ABORT response from some repository, it returns false; if it receives a CONFLICT response, it retries the transaction with a new TID, after waiting a random backoff.

4.3 Single-Repository Transactions

The basic protocol for single-repository transactions has much in common with how existing single-node storage systems work. The protocol timeline is shown in Figure 5.

The protocol for read-write transactions is as follows:

1. When a repository receives a client REQUEST it assigns it a timestamp that is greater than the highTS sent by the client, the timestamp of the most recently executed transaction at the repository, and the current clock value.

2. The repository performs a stable log write to record both the request and the assigned timestamp, so that this information will persist across failures.

3. The transaction is now ready to be executed. Transactions are executed in timestamp order, by making

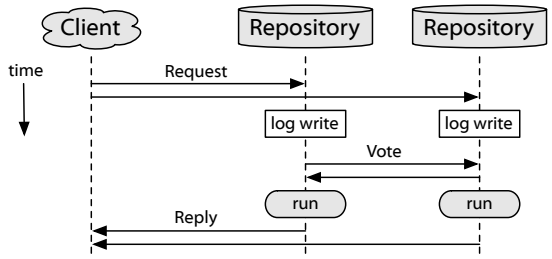


Figure 6: Timeline for independent transactions.

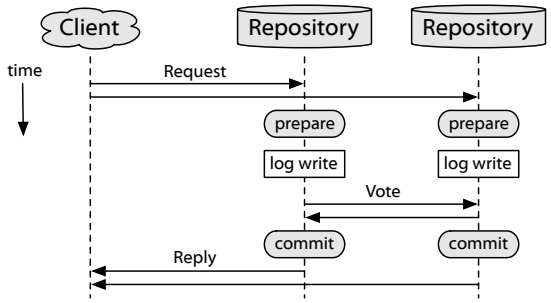


Figure 7: Timeline for coordinated transactions.

a `run` upcall to the application. Once a transaction is executed, a COMMIT reply containing the result of the upcall is sent to the client.

Additional transactions can be processed while awaiting completion of a stable log write; these requests will be executed in timestamp order.

The protocol for read-only transactions is the same as for read-write transactions, except that a stable log write is not required in Step 2 of the protocol. Since read-only transactions do not modify the service state, they can be retried in the case of failure.

4.4 Independent Distributed Transactions

Independent transactions are ordered with respect to all other transactions without locking or conflicts. Granola achieves this by executing each independent transaction at the same timestamp at all transaction participants. We determine the timestamp by using a distributed voting mechanism. Each participant nominates a proposed timestamp for the transaction, the participants exchange these nominations in VOTE messages, and the transaction is assigned the highest timestamp from among these votes.

The protocol for transactions that modify data is as follows; the timeline is shown in Figure 6.

1. The repository selects a *proposed* timestamp for the transaction that is higher than highTS (sent by the client), the timestamp of the most recently executed transaction at the repository, and the current clock value.

2. The transaction request and timestamp proposal are recorded using a stable log write.

3. The repository sends a COMMIT VOTE message containing the proposed timestamp to the other participants. The repository can process other transactions after the vote has been sent.

4. The repository waits for votes from other participants. If it receives a CONFLICT vote, it ceases processing the transaction and sends this response to the client. A conflict vote will be received only from a participant that is operating in locking mode, as described in the next section.

5. Once the repository receives commit responses from all other participants, it assigns the transaction the *highest* timestamp from the votes; this timestamp will be consistent across all participants. The transaction is now ready to be executed.

6. The transaction is executed at the assigned timestamp, in timestamp order, and a reply is sent to the client.

The protocol for read-only transactions is similar, except that the stable log write is not required. Since these transactions do not modify the service state, the client proxy can retry a read-only transaction if a participant fails while executing the protocol.

As mentioned, the repository can process other transactions while waiting for votes. In all cases we, guarantee serializability by executing in *timestamp* order. A transaction won't be executed until after any concurrent transaction with a lower timestamp at the repository. It is thus possible for execution of a transaction to be delayed if a transaction with a lower timestamp has not yet received a full set of votes. Longer-term delays can occur if a transaction participant has failed; recovery from a failed participant is discussed in Section 5.4.

4.5 Coordinated Distributed Transactions

We now describe the protocol used for coordinated transactions, and the impact on single-repository and independent transactions when in locking mode. Coordinated transactions require participants to agree whether to commit or abort. They require locking to support concurrency; otherwise, one transaction might modify state that was used by a concurrent transaction to determine its vote.

The protocol for coordinated transactions is as follows; the timeline is shown in Figure 7.

1. Coordinated transactions first undergo a prepare phase. This is accomplished by issuing a `prepare`

upcall to the application. The application acquires any locks required by the transaction, and returns its vote, holding the locks only if it decides to commit.

2. The repository selects a timestamp for the transaction that is greater than highTS, the timestamp of the last executed transaction and the current clock value.

3. The transaction request, vote, and proposed timestamp are recorded using a stable log write.

4. The repository sends its VOTE along with the proposed timestamp. If the vote is ABORT or CONFLICT, it immediately returns this result to the client, and ceases processing the transaction.

5. The repository waits for votes from other participants. If it receives an ABORT or CONFLICT vote it makes an `abort` upcall to the application which releases locks and reverts any changes, sends the ABORT or CONFLICT REPLY to the client, then ceases processing the transaction.

6. Once the repository has received COMMIT votes from all other participants, it assigns the transaction the highest timestamp from the votes, and immediately executes the transaction by issuing a `commit` upcall to the application. The application releases any locks, and returns the transaction result. The repository can then send a COMMIT REPLY to the client.

Locking Mode. The protocol for single-repository and independent transactions is different in locking mode. Independent transactions are processed using the coordinated transaction protocol with `prepare` and `commit` upcalls. The client proxy will retry an independent transaction if it receives a CONFLICT response; it will never receive an ABORT response for an independent transaction.

We avoid holding locks for single-repository transactions by attempting to execute them as soon as they have been assigned a timestamp, but before the transaction is logged, by using a `run` upcall. The application checks locks for a `run` upcall if issued concurrently with other distributed transactions. If the `run` upcall is successful, the repository issues a stable log write to record the timestamp for the transaction before responding to the client. If there is a lock conflict, the application returns `false`, and the repository responds to the client with a CONFLICT response then discards the transaction. No log write is required for the aborted transaction.

The protocol for read-only single-repository transactions is the same, except that the stable log write is not required. The response to the client must be buffered until the most recent stable log write is complete, to avoid externalizing the effects of any previous transaction that has not yet been logged. We discuss the recovery implications of executing a transaction before logging in Section 5.2.

Switching Modes. There may be single-repository and independent transactions in progress when a coordinated transaction arrives while the repository is in timestamp mode. We handle this situation by issuing `prepare` upcalls for the independent transactions, and following the locking protocol. If all prepares succeed, we go on to process the coordinated transaction; otherwise, we block the coordinated transaction and remain in timestamp mode until all earlier prepares succeed.

It is desirable to switch back to timestamp mode as soon as possible, to avoid the cost of locking. Once all coordinated transactions have completed, the repository issues an `abort` upcall for any current independent transactions. This releases their locks, and allows the repository to transition into timestamp mode. The independent transactions will be executed using `run` upcalls in timestamp order, once their final timestamps are known.

4.6 Consistency

Granola provides serializability for transactions. In timestamp mode our consistency model is straightforward: Timestamps define a *total ordering* of transactions, guaranteeing serializability; all participants observe a transaction to execute at a single common timestamp. Clients propagate timestamps between repositories, to ensure that a transaction is not assigned a timestamp lower than any transaction that may have preceded it. Locking is not required when in timestamp mode, since each transaction executes serially with a single application upcall at each participant.

Granola allows each repository to execute its part of an independent transaction without knowing what is happening at the other participants: it only knows that they will ultimately select the same timestamp. This means that it is possible for a client to observe the effect of a distributed transaction T at one repository before another participant has executed it. Since any subsequent request from the client will carry a highTS value at least as high as T, however, we can guarantee that this request will be delayed if necessary and execute after T at any participant.

Locks are required when handling coordinated transactions, to ensure that a transaction does not observe or invalidate the state used to determine the commit or abort vote for a concurrent transaction. Repositories may execute transactions out of timestamp order when in locking mode, since locking is sufficient to guarantee serializability [16]. Timestamps thus may not represent the *commit* order of transactions, but still represent a valid *serial* order, since any transactions that execute out of timestamp order are guaranteed not to conflict.

Our transaction protocol does not provide *external* consistency [23], meaning that consistency is not guaranteed when communication occurs outside the system. Granola relies on clients including their highTS value on each request, which will not be included on out-of-band communication. External consistency can be provided for client-to-client communication if these messages include the highTS value of the sender. While violations of external consistency are possible for communication that occurs completely outside the system, such violations are unlikely since they are only possible within a small window of time, proportional to the clock skew between repositories [13].

5 Failures and Recovery

This section discusses the mechanisms used to handle individual node failures. We also discuss what happens when problems such as a network partition cause repositories to become unresponsive for an extended period of time.

5.1 Replication

Granola requires that stable log writes remain durable, and that repositories recover quickly from failure. We accomplish this using state machine replication [30]; while disk writes could have been used for durability, they do not provide fast recovery from failure. Our replication protocol uses an improved version of Viewstamped Replication [24, 28]; Paxos [22] provides an equivalent consensus protocol and could also have been used.

Repositories are replicated using a set of $2f+1$ replicas to survive f crash failures. One replica is designated the *primary*, and carries out the Granola protocol. *Backup* replicas carry out a *view change* to select a new primary if the old one appears to have failed.

Stable log writes involve the primary sending the log message to the backups and waiting for f replies, at which point the log is stable. The primary does not stall while waiting for replies; it continues processing incoming requests in the meantime. The replication protocol uses batching of groups of log messages [10] to reduce the number of messages sent to backups.

Each log message records the request, proposed timestamp and vote for each transaction. The primary piggybacks the final timestamp assigned to each transaction on subsequent log messages, to facilitate pruning the log at the backups. The backups execute transactions in timestamp order once the final timestamp is known.

Our protocol does not require disk writes as part of the stable log write, since we assume that replicas are failure-independent. Information is written to disk in the background, e.g., as required by main memory limitations. Failure-independence can be achieved by locating replicas in disjoint locations, or by equipping them with battery-backed RAM.

5.2 Repository Recovery

This section discusses issues that arise due to failovers.

- When a failover occurs, it's possible that the old primary does not know it has been superseded. This can be a problem for read-only single-repository transactions since they may observe stale data if run at the old primary. We avoid this problem by using the leases mechanism introduced in Harp [25], which guarantees there is only ever a single primary.

- The new primary might not know about recent read-only independent transactions for which the old primary sent votes. The new primary may thus receive such a request and select a different timestamp, and as a result the client proxy may receive replies with different timestamps. In this case the client proxy discards the replies and reissues the transaction with a higher TID.

- The new primary will know all votes sent by the previous primary for distributed read/write transactions, but may not know the final timestamp assigned to some recently completed transactions. The new primary recovers this information by resending its votes to solicit votes it is missing.

- In locking mode, the new primary may execute transactions in a different order than the old primary since the execution order in locking mode depends on the order in which votes are received. Both the old and new primaries will only execute transactions in an order consistent with the timestamp order, however, and hence will not deviate in the serial ordering.

- In locking mode, single-repository transactions are executed before being logged and externalized, and thus may not persist into the new view. In this case the previous execution of the transaction is forgotten. When the old primary recovers, it must undo the execution of any such transactions; this can be accomplished by reverting to a checkpoint and replaying transactions from the log.

5.3 Client Recovery

Clients can ensure uniqueness of sequence numbers (which comprise TIDs) by writing periodic sequence number ranges to disk, and only using sequence numbers within each range. Alternatively, a client can recover its timestamp after failure by synchronizing its clock and

ensuring that a period of time equal to the maximum expected clock skew has elapsed. After this it can adopt its clock value as its new latest-observed timestamp, which will be at least as high as the timestamp it knew before it failed. Note that here we are depending on synchronized clocks for correctness, where otherwise we have not needed this assumption.

If there are no explicit consistency constraints between client sessions, the client proxy can instead set its latest-observed timestamp to 0 when recovering from failure.

5.4 Long-term Failures

Since Granola provides strong consistency, we may have to stall because of failure. Replication masks the failure of individual nodes, but cannot provide availability if an entire replica group becomes unavailable. Permanent loss of a replicated repository may result in data loss and will likely require human intervention. This section describes how the rest of the system can make progress after failure of a replicated repository, in particular the situation where a non-failed repository is unable to obtain votes from a failed or unresponsive participant.

When a repository notices that a participant is unresponsive, it first attempts to resolve the transaction by resending its votes to any other participants; these participants will respond with the transaction status if they completed the transaction. If this is unsuccessful after a timeout, however, we proceed as follows.

Locking Mode. The repository needs to hold locks for *incomplete* distributed transactions that are awaiting votes from the failed participant until the participant recovers; subsequent transactions will be sent a CONFLICT REPLY if they conflict with these locks. The repository periodically resends each vote, and will commit or abort an incomplete transaction if notified by another participant that knows the transaction outcome.

Timestamp Mode. Recovery is more complicated if the repository is in timestamp mode, since it will not have acquired locks for the incomplete transactions, and must thus execute transactions in timestamp order. This results in a set of *blocked* distributed transactions, for which the repository has a full set of votes, but cannot yet execute because they are queued behind an *incomplete* transaction that currently has a lower timestamp based on votes so far. Single-repository transactions and read-only independent transactions are not included in these sets, since they can be sent a CONFLICT REPLY without causing repository state to diverge.

The repository first attempts to transition into locking mode by issuing a `prepare` upcall for each blocked transaction, in the current timestamp order, stopping if there is a lock conflict. If the `prepare` is successful for an incomplete transaction, the repository will continue holding the locks. If the `prepare` is successful for a blocked transaction, the repository executes it immediately by issuing a `commit` upcall, responds to the client, and removes it from the queue; this is safe because any request later in the queue will end up with a later timestamp and run after it, and this request doesn't conflict with any requests earlier in the queue.

If the repository is able to prepare all transactions without any lock conflicts, then it transitions into locking mode and follows the locking mode recovery protocol above. If it finds a conflict, however, it must cease preparing the remaining transactions. In the absence of conflicts, the repository can acquire locks for transactions in any order, since none of the transactions interfere, but if there is a lock conflict, the locks acquired for a transaction may depend on the order in which it is run. For example, consider a transaction that reads a stock level from one database table, then updates one of two different tuples depending on whether or not the stock is above a certain level; in this case the lockset depends on the relative ordering of transactions that modify the level.

We acquire locks in an *order-independent* way by requiring applications to implement a `forcePrepare` upcall. This function acquires locks the transaction might acquire in *any* execution order. In our stock level example, the application would acquire locks on both tuples, regardless of the stock level. For many applications, transactions are already order-independent, and this superset is the same as the set of locks acquired by a regular `prepare` upcall, particularly for transactions that update fields that are known ahead of time [7]. In other applications it might involve escalating lock granularity, e.g., from tuple-level locks to sets of locks or even to table locks.

The `forcePrepare` upcall acquires all locks even if there is a lock conflict, to allow the repository to accumulate locks for all blocked and incomplete transactions.

The repository iterates through the incomplete and blocked transactions (excluding blocked transactions that were completed while performing the `prepare` upcalls) and issues `forcePrepare` upcalls for them in the current timestamp order. The upcall returns true if there is no lock conflict; in the case of a blocked transaction this means it does not conflict with any transaction possibly ahead of it in the final serial order, and the repository can execute it, thus releasing its locks, and respond to the client.

Once locks have been acquired for all blocked and incomplete transactions, the repository can transition into locking mode and resume accepting new transactions.

6 Evaluation

We implemented Granola in Java, and deployed it on 10 2005-vintage 3.2 GHz Xeon servers with 2 GB RAM, with a cluster of 10 more powerful 2.5 GHz Core2 Quad servers with 4 GB RAM to provide client load. These machines were connected by a gigabit LAN with a network latency of ~ 0.2 ms. Multiple clients are colocated on a given machine to fully load the repositories, and wide-area network delay is emulated by delaying outgoing packets in our network library. Replication is emulated by running the protocol locally with appropriate network delay. We use this platform to examine Granola's scalability, latency, and resistance to lock conflicts and overhead.

We compare Granola's performance against the transaction coordination protocol from Sinfonia [7], which uses a more traditional lock-based version of two-phase commit. We use this coordination protocol by having clients issue single-repository transactions to the repositories (memory nodes) and issue distributed transactions to a single coordinator (application node). We used a single high-powered 16-core 1.6 GHz Xeon server with 8 GB RAM to serve as the coordinator, to avoid aborts due to contention from multiple masters.

Our implementation of Sinfonia differs in that we don't require clients to explicitly provide the read/write locksets for a transaction, and instead allow general operations through the use of `prepare` and `commit` upcalls. While we refer to this implementation as Sinfonia in our benchmarks, it could be used to represent any similarly-efficient implementation of two-phase commit. No locking is performed for single-repository transactions when running in isolation; the implementation must check whether a single-repository transaction conflicts with any active locks when running concurrently with distributed transactions.

We first examine the performance characteristics of Granola on a set of microbenchmarks, followed by an evaluation on a more complex transaction processing benchmark. Our figures show 95% confidence intervals for all datapoints.

6.1 Micro-benchmarks

Our micro-benchmarks examine a counter service. Each update modifies either a counter on a single repository, or counters distributed across multiple repositories. We vary the conflict rate for coordinated transactions by adjusting the ratio of conflicting updates issued by a client. Since the protocol for read-only operations is similar to many other distributed storage systems, these benchmarks focus exclusively on read/write transactions.

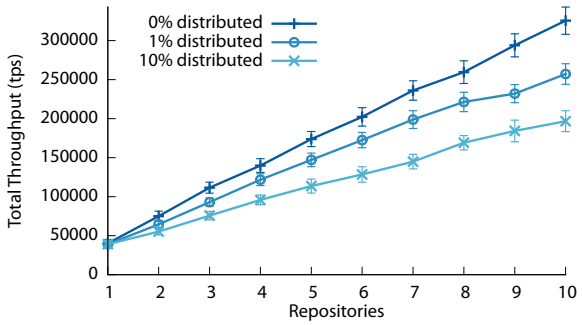


Figure 8: Throughput scalability.

6.1.1 Base Performance

We measured a maximum throughput of approximately 50,000 tps (transactions/sec) with a per-transaction latency of under 1 ms, increasing to 65,000 tps with under 10 ms latency, for single-repository transactions on a single compute core. Throughput was CPU-bound, and we observed more than twice this throughput on a more powerful machine. We also examined a wide-area configuration with approximately 10 ms emulated one-way delay between replicas, which resulted in similar throughput and a baseline per-transaction latency of approximately 22 ms. Wide-area replication did not impose a throughput penalty despite additional per-request latency, since our replication protocol can handle transactions in parallel.

6.1.2 Scalability

We illustrate Granola's scalability in Figure 8. Clients issue each request to a random repository, with between 0 and 10% of requests issued as distributed 2-repository independent transactions. This figure shows a local-area replication configuration. Configurations with 10 ms one-way delay between repositories and between replicas gave similar throughput but required significantly more clients to load the system, due to higher request latency.

There is a slight drop in per-repository throughput when moving from one repository to multiple repositories, due to the overhead of buffering transactions with higher `highTS` values, but this stabilizes at higher numbers of repositories. This occurs in our microbenchmark since clients and repositories are colocated on the same LAN, and hence the clock skew can be greater than the latency between requests; this effect is less likely when there is a network delay between clients and repositories.

Per-client throughput is lower for distributed transactions, owing to the additional communication delay; distributed-transaction latency was found to be consistently twice the latency of single-repository transactions.

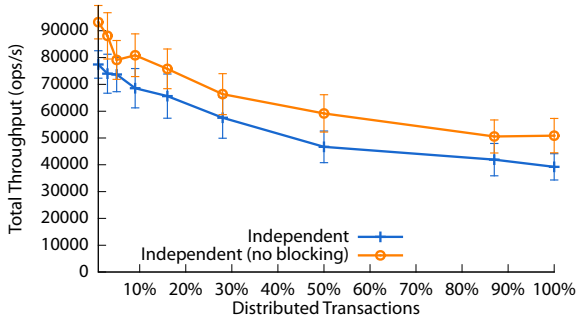


Figure 9: Throughput with distributed transactions.

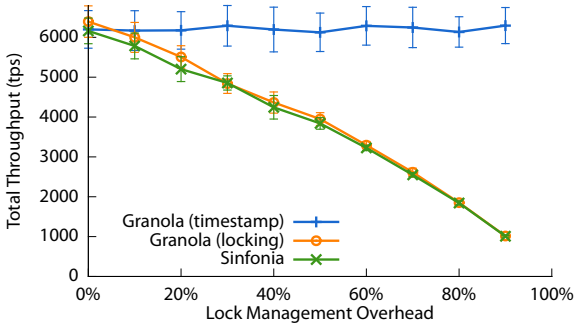


Figure 10: Throughput with locking/logging overhead.

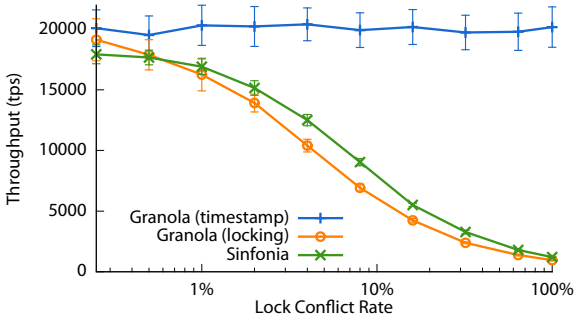


Figure 11: Throughput with varying lock conflict rate.

6.1.3 Distributed Transactions

Figure 9 shows the impact of workloads with between 0 and 100% distributed transactions. For each data-point, we examine workloads composed of independent transactions concurrently with single-repository transactions in timestamp mode; we examine locking mode in subsequent sections. We compare the overhead of the timestamp protocol against a version of Granola that never blocks a transaction to wait for earlier timestamps to complete, and thus does not provide consistency.

This figure shows throughput in terms of *operations* per second, since each distributed transaction involves running an operation on each repository. An optimal coordination scheme would exhibit constant throughput, independent of the fraction of distributed transactions. Granola scales well with an increase in distributed transactions, but we observe throughput less than this optimal value, due to communication and processing overhead. We observe a 10–20% reduction in throughput from waiting for earlier transactions to complete, due to timestamp constraints. We examine distributed transaction throughput on a more realistic workload in Section 6.2.

6.1.4 Locking

Locking introduces overhead in two key ways: the execution cost of acquiring locks and recording undo logs; and the wasted work from having to retry transactions that abort due to lock conflicts. We examine both these components separately in the following two sections, and examine locking on a realistic workload in Section 6.2.

Lock Management Overhead. Figure 10 shows the throughput of Granola and our version of Sinfonia on a two-repository topology as a function of lock management cost. We examine a workload composed entirely of distributed transactions, and run Granola in both timestamp mode and locking mode. This experiment measures lock overhead with no lock contention; we set transaction execution cost to be equivalent to the cost of a `new_order` transaction in TPC-C, and vary the lock overhead as a fraction of this execution cost.

As expected, Granola in timestamp mode is unaffected by locking overhead, while both Sinfonia and Granola in locking mode drop in throughput as lock management overhead increases. Typical values for lock management cost in OLTP workloads are in the vicinity of 30–40% of total CPU load [18, 19]. At this level Granola gives 25–50% higher throughput than lock-based protocols, even in the absence of lock contention.

Lock Contention. We investigate the impact of lock *contention* in Figure 11, on a two-repository topology with 100% distributed transactions. We control the lock conflict rate by having transactions modify either a private or shared counter. This experiment measures lock contention in isolation, and we tailor our application to have negligible lock management overhead.

Throughput for Sinfonia and Granola in locking mode deteriorates fairly rapidly as lock conflict rate increases, due to the cost of retrying aborted transactions; note the logarithmic x-axis. Throughput for these transactions drops to approximately 1000 tps at 100% contention, which correlates with the average 1 ms latency we observed for each non-aborted transaction.

Unlike our other experiments, Sinfonia's maximum throughput at low contention is limited here by our use

of a single coordinator, which serves as a bottleneck. Sinfonia achieves slightly higher throughput than Granola in locking mode when there is *high* contention, since the coordinator presents a consistent transaction ordering to repositories, unlike in Granola where transactions may be received by repositories in conflicting orders. We evaluated an extension to Granola to support the use of lightweight coordinator nodes, but observed minimal benefit in typical workloads [13].

6.2 Transaction Processing Benchmark

We evaluate performance on an application based on the TPC-C transaction processing benchmark [6]. This benchmark models a large order-processing workload, with complex queries distributed across multiple repositories. Our implementation stores the TPC-C dataset in-memory and executes transactions as single-round stored procedures.

We used the C++ implementation of TPC-C from the H-Store project [32] for our client and server application code.[2] The codebase that we used was designed for a single node deployment and had no explicit support for distributed transactions. By interposing the Granola platform between the TPC-C client and server, we were able to build a scalable distributed database with minimal code changes; code modifications were constrained to calling the Java `ClientProxy` from the C++ client, responding to transaction requests from the GranolaApplication server, and translating warehouse numbers to repository IDs.

We adopt the data partitioning strategy proposed in H-Store. This partitioning ensures that all transactions can be expressed as either single-repository or independent transactions. We were able to disable locking and undo logging when evaluating Granola, since TPC-C involves no coordinated transactions. We also compare Granola and Sinfonia against a version of Granola that is set to always operate in locking mode, to measure the impact of lock-based concurrency control.

Scalability. We examine scalability in Figure 12. This experiment uses a single TPC-C warehouse per repository, and increases the number of clients to maximize throughput. 10.7% of transactions in this benchmark are issued to multiple participants.

All systems exhibit the same throughput in a single-repository configuration, since they both have similar overhead in the absence of locking or distributed transactions. Throughput drops for the lock-based protocols on multiple nodes, however. The TPC-C implementation is highly optimized and executes transactions efficiently, hence the lock overhead imposes a significant relative penalty; the overhead of locking and allocating undo records was approximately equal to the cost of executing each operation, in line with similar measurements on the same workload [20]. Throughput reduction is also heavily impacted by the cost of retrying transactions that abort due to lock conflicts. Sinfonia offers slightly lower performance than Granola in locking mode, due to the additional overhead and latency of communicating with the coordinator.

[2]This implementation does not *strictly* adhere to the TPC-C spec., e.g., does not implement client "wait times" between requests [19].

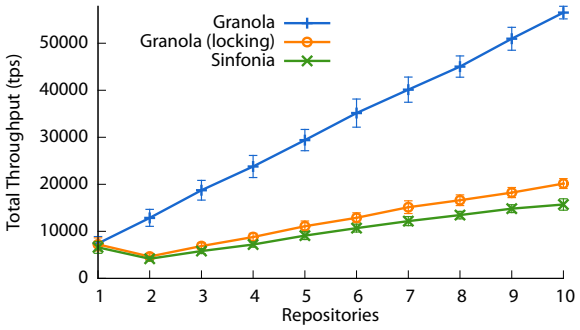


Figure 12: Scalability of TPC-C implementation.

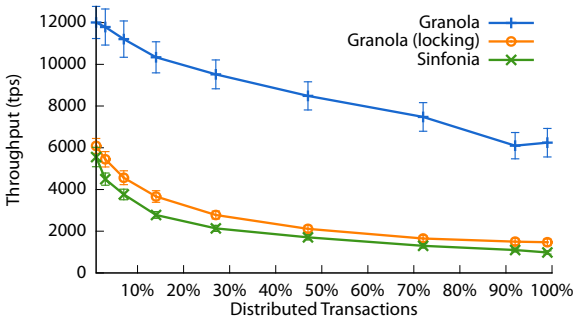


Figure 13: TPC-C throughput `new_order` transactions.

We observe a relatively constant latency regardless of system size, with average distributed transaction latencies of 2.9, 3.2, and 4.9 ms for Granola, Granola (locking) and Sinfonia respectively. Sinfonia encounters higher latency due to the additional communication delay.

Distributed Transactions. We further examine coordination overhead by modifying TPC-C to vary the proportion of distributed transactions. This workload is composed entirely of `new_order` transactions and we adjust the likelihood that an item in the order will come from a remote warehouse [20]. The results for this benchmark are shown in Figure 13, for a two warehouse configuration on two repositories.

These results echo the previous benchmark, with the performance difference dominated by lock conflicts and lock management overhead. Granola achieves better resilience to distributed transactions in this benchmark than in our microbenchmarks, since overhead in TPC-C is dominated by transaction execution costs rather than proto-

col effects. Throughput for Granola in timestamp mode decreases by approximately 50% when moving from 0 to 100% distributed transactions. This represents a very low performance penalty for distributed transactions, since each given distributed transaction involves execution cost on two repositories instead of one.

7 Related Work

There has been a long history of research in transactional distributed storage, including a rich literature on distributed databases [9, 15, 27]. These systems provide support for interactive transactions, whereas Granola targets a simpler single-round transaction model that can nonetheless be used to support a wide number of applications [7, 32].

Relaxed Consistency. Many systems [17, 21, 29, 34] relax the consistency guarantees provided by traditional databases, in order to provide increased scalability and resilience to network or hardware partitions. The growth of cloud computing has led to a resurgence in popularity of large-scale storage systems with weaker consistency, typified by Amazon's eventual-consistency Dynamo [14] and a wealth of others [2–5, 12]. These systems target high availability and aim to be "always writable", but sacrifice consistency and typically offer constrained transaction interfaces, such as Dynamo's read/write distributed hash table interface.

Per-Row Consistency. Systems such as SimpleDB [1] and Bigtable [11] provide consistency within a single row or data partition, but do not provide ACID guarantees between these entities. A significant downside to relaxed-consistency storage systems is the complicated application semantics presented to clients and developers when operating with multiple data items. More recent protocols such as COPS [26] and Walter [31] attempt to simplify application development by providing stronger consistency models: *causal+* and *parallel snapshot isolation* respectively. These models do not prevent consistency anomalies however, and require the developer to reason carefully about the correctness of their application.

Strong Consistency. Megastore [8] represents a departure from the traditional wisdom that it's infeasible for large-scale storage systems to provide strong consistency. Megastore is designed to scale very widely, uses state-machine replication for storage nodes, and offers transactional ACID guarantees. As in SimpleDB [1], Megastore ordinarily provides ACID guarantees within a single entity group, but also supports the use of standard two-phase commit to provide strong consistency between groups. CRAQ [33] primarily targets consistency for single-object updates, but mentions that a two-phase commit protocol could be used to provide multi-object updates. Granola is more heavily optimized for transactions that span multiple partitions, and provides a more general operation model.

Granola is most similar in design to Sinfonia [7]. Sinfonia also supports reliable distributed storage with strong consistency over large numbers of nodes. Sinfonia's *mini-transactions* express transactions in terms of read, write and predicate sets, whereas Granola supports arbitrary operations and does not require a priori knowledge of lock-sets. Granola also provides fewer message delays in the transaction coordination protocol. The most significant difference is Granola's support for independent distributed transactions; this is of considerable benefit in suitable workloads, avoiding conflicts and lock overhead.

The H-Store project argues for the relevance of transactions that fit the independent model, and observes that these transactions can be handled without locking [32]. The protocol sketched in the position paper was not a full distributed implementation however, and does not work with failures, delays and clock skew; later complete implementations used different techniques and did not optimize for independent transactions [20, 36]. Granola introduces a novel transaction coordination protocol based on timestamp exchange to provide the first complete protocol that supports independent transactions in a distributed setting.

The Calvin transaction coordination protocol [35] was developed in parallel with Granola, and provides similar functionality. Rather than using a distributed timestamp voting scheme to determine execution order, Calvin delays read/write transactions and runs a global agreement protocol to produce a deterministic locking order.

8 Conclusion

Granola is a distributed transaction infrastructure that provides serializability for one-round transactions. This simplifies the reasoning required by application developers and users of the system. Granola also supports a general operation model, which allows development of arbitrary storage applications.

Granola implements new protocols that provide lower overhead for transaction coordination than in previous work. Distributed transactions complete with one stable log write and only three message delays.

Most significantly, this paper introduces explicit support for independent distributed transactions. Independent transactions appear in many common workloads and allow the development of high-performance distributed applications. Granola uses a timestamp-based implementation of independent transactions that provides a substantial performance benefit, due to an absence of lock conflicts and management overhead.

Acknowledgments

We thank Dan Ports, the anonymous reviewers, and our shepherd, Jon Howell, for their helpful feedback. This research was supported under NSF grant CNS-0834239.

References

[1] Amazon SimpleDB. http://aws.amazon.com/simpledb/.

[2] Apache Cassandra. http://cassandra.apache.org.

[3] Apache CouchDB. http://couchdb.apache.org.

[4] Apache HBase. http://hbase.apache.org.

[5] MongoDB. http://www.mongodb.com.

[6] TPC benchmark C. Technical report, Transaction Processing Performance Council, February 2010. Revision 5.11.

[7] M. K. Aguilera, A. Merchant, M. A. Shah, A. C. Veitch, and C. T. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM TOCS*, 2009.

[8] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J. M. Lon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.

[9] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, a highly parallel database system. *IEEE TKDE*, March 1990.

[10] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM TOCS*, 2002.

[11] 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 *OSDI*, 2006.

[12] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *VLDB*, 2008.

[13] J. Cowling. *Low-Overhead Distributed Transaction Coordination*. PhD thesis, MIT, June 2012.

[14] 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 *SOSP*, 2007.

[15] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H. l Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE TKDR*, March 1990.

[16] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *CACM*, Nov. 1976.

[17] R. Guy, J. Heidemann, W. Mak, T. Page Jr., G. Popek, and D. Rothneier. Implementation of the Ficus replicated file system. In *USENIX*, 1990.

[18] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, 2008.

[19] E. P. C. Jones. *Fault-Tolerant Distributed Transactions for Partitioned OLTP Databases*. PhD thesis, MIT, 2012.

[20] E. P. C. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*, June 2010.

[21] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing High Availability Using Lazy Replication. *ACM TOCS*, Nov. 1992.

[22] L. Lamport. The Part-Time Parliament. Technical Report Research Report 49, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, Sept. 1989.

[23] K.-J. Lin. Consistency issues in real-time database systems. In *System Sciences*, 1989.

[24] B. Liskov and J. Cowling. Viewstamped replication revisited. Technical report, MIT CSAIL, Cambridge, MA, 2012.

[25] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp File System. In *SOSP*, 1991.

[26] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.

[27] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R* distributed database management system. *ACM TODS*, December 1986.

[28] B. Oki and B. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *PODC*, 1988.

[29] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, David, and C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE TC*, 1990.

[30] F. B. Schneider. The state machine approach: A Tutorial. Technical Report TR 86-600, Cornell University, Dept. of Computer Science, Ithaca, N. Y., Dec. 1986.

[31] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, 2011.

[32] M. Stonebraker, S. R. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *VLDB*, 2007.

[33] J. Terrace and M. J. Freedman. Object storage on CRAQ: high-throughput chain replication for read-mostly workloads. In *USENIX ATC*, 2009.

[34] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, and M. J. Spreitzer. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *SOSP*, 1995.

[35] A. Thomson, T. Diamond, S. chun Weng, P. Shao, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.

[36] VoltDB Inc. VoltDB. http://voltdb.com.

High Performance Vehicular Connectivity with Opportunistic Erasure Coding

Ratul Mahajan Jitendra Padhye Sharad Agarwal Brian Zill

Microsoft Research

Abstract — Motivated by poor network connectivity from moving vehicles, we develop a new loss recovery method called opportunistic erasure coding (OEC). Unlike existing erasure coding methods, which are oblivious to the level of spare capacity along a path, OEC transmits coded packets only during instantaneous openings in a path's spare capacity. This transmission strategy ensures that coded packets provide as much protection as the level of spare capacity allows, without delaying or stealing capacity from data packets. OEC uses a novel encoding that greedily maximizes the amount of new data recovered by the receiver with each coded packet. We design and implement a system called PluriBus that uses OEC in the vehicular context. We deploy it on two buses for two months and show that PluriBus reduces the mean flow completion time by a factor of 4 for a realistic workload. We also show that OEC outperforms existing loss recovery methods in a range of lossy environments.

1. Introduction

Internet access on-board buses, trains, and ferries is increasingly common. Many public transit agencies provide this access today [48, 46, 14]. It is seen as an added amenity that boosts ridership, even in the age of the 3G smart phones [28, 35]. Corporations also provide such access on their commute vehicles [45, 47]. For instance, over one-quarter of Google's employees in the Bay Area use such connected buses [45]. By all accounts, riders greatly value this connectivity.

Our work is motivated by our experiences of poor performance of such connectivity and those of other users [43, 44]. Experiences with its commuter service led Microsoft to pre-emptively warn the riders that "there can be lapses in the backhaul coverage or system congestion" and suggest "cancel a failed download and re-try in approximately 5 minutes." Despite increasing popularity and a unique operating environment, the research community has paid little attention to how to best engineer these networks.

Figure 1 shows the typical way to enable Internet access on buses today. Riders use WiFi to connect to a device on the bus (e.g., [13]), which we call *VanProxy*. The device provides Internet access using one or more links based on wide-area wireless network (WWAN) technologies such as EVDO or HSDPA. The key to application performance in this setup is the quality of connectivity provided by the WWAN links.

To understand this connectivity, we conducted detailed measurements of multiple technologies. Consistent with earlier findings [34, 18], we find that WWAN paths offer poor service from moving vehicles. They have high delays and frequently drop packets. Occasionally, they suffer "blackout" periods with very high loss rates. Thus, poor application performance is only to be expected.

To improve user experience, we must mask losses from applications and offer them a more reliable communication channel. While numerous loss recovery schemes exist, we find that they fall short in this environment. Existing schemes can be categorized as either retransmission-based (ARQ) and erasure coding based. Retransmission-based schemes perform poorly because of the high delay in receiving feedback from the receiver.

Proactive erasure coding is more appropriate in high-delay scenarios but existing schemes (e.g., Maelstrom [2], CORE [23], LT-TCP [41]) have a basic limitation: they are oblivious to spare capacity along a path. For a given set of data packets, the number of erasure coded packets sent does not depend on the available capacity of the path. If this coding redundancy is low, existing schemes do not provide sufficient protection even though there may be spare capacity in the system. If it is high, valuable capacity is stolen from data packets.

In this paper, we explore a new point in the design space of erasure coding and evaluate it in the vehicular context. Our method, called opportunistic erasure coding (OEC), dynamically adjusts coding redundancy to match the spare capacity of the path at short time scales. Matching at short time scales is important because, as we show, the traffic is highly bursty. Matching coding overhead to average spare capacity is not sufficient, as it can lead to significant short-term mismatches.

To match coding redundancy to spare capacity at short-time scales, OEC sends coded packets opportunistically, based on an estimate of bottleneck queue length. Coded packets are transmitted as soon as and only when the queue is deemed empty. Thus, OEC does not delay data packets and yet provides as much protection as available capacity allows.

To make the best use of such opportunistic transmissions, we construct coded packets using a greedy encoding that maximizes the expected number of data packets recovered using each coded packet. Our encoding can be considered a generalization of Growth codes [19] that

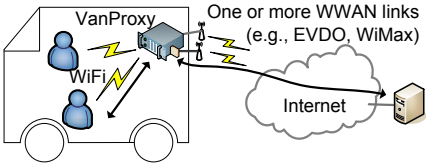


Figure 1: A common way of providing Internet access on board vehicles today.

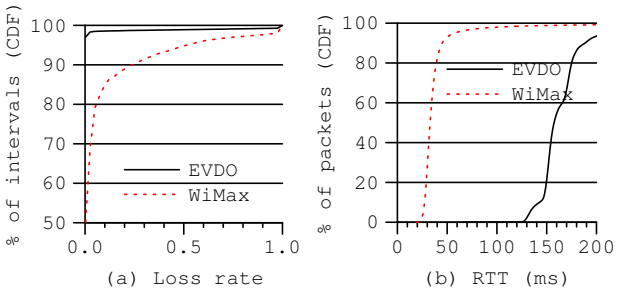


Figure 2: (a) Loss rate for paths to the buses. The y-axis begins at 50%. (b) Path round trip times (RTT).

explicitly accounts for the information available at the receiver while constructing the next coded packet. In contrast, conventional erasure coding methods such as Reed-Solomon [33] or LT [24] aim to minimize the number of packets needed at the receiver to recover all data. But when the required threshold of packets are not received, they recover very little data [36]. In a highly dynamic environment, it is difficult to guarantee that the required threshold number of packets will be sent, let alone received. Thus, these codes are not suitable for our use.

The combination of opportunistic transmissions and our encoding means that OEC greedily maximizes goodput with each packet transmission. OEC retains this property even when data is spread across multiple paths with disparate loss and delay. We accomplish this through delay-based path selection [9]: each data packet is sent over the path that is estimated to deliver it first.

We design and implement a system called PluriBus that applies OEC to the vehicular context. We deploy PluriBus on two buses for two months. Each bus is equipped with two WWAN links, one EVDO and one WiMax.

Our evaluation using this deployment as well as controlled experiments show that PluriBus is highly effective over a range of network conditions. In our deployment, it reduces the mean flow completion time for a realistic workload by a factor of 4 compared to the current practice of not using any loss recovery method (beyond end-to-end TCP). Compared to using retransmissions or capacity-oblivious erasure coding, OEC reduces the mean flow completion time by at least a factor of 1.4.

2. Target environment

We begin by characterizing the network and workload in our target environment. To understand the connectivity provided by WWAN links to moving vehicles, we use two buses that ply around the Microsoft campus from 7 AM to 7 PM on weekdays. Each bus has a computer equipped with an 1xEVDO (Rev. A) NIC on Sprint's network and a (draft standard) WiMax modem on Clearwire's network.

2.1 Network path characteristics

We characterize path quality by sending packets between the bus and a computer connected to the wired Internet. A packet is sent along each provider in each direction every 100 ms. Our analysis is based on two weeks of data. Figure 2(a) shows the CDF of loss rates, averaged over 5 second intervals, from the wired host to the buses. The reverse direction has similar behavior. We see that both paths are lossy. WiMax is worse—half of the intervals experience some packet loss, and 15% suffer over 10% loss. For EVDO, 97% of the intervals see no loss, but 2% suffer over 10% loss. Note that these losses are measured at the IP layer and represent cases where low-level reliability mechanisms (e.g., link-layer FEC) have failed. They will be experienced by TCP connections traversing these links.

Our observations are consistent with other WWAN studies [22, 18]. These studies also find that most losses are not due to congestion but occur due to problems inherent in wireless transmissions. Wireless collisions with other WWAN clients are not an issue; unlike WiFi, the WWAN MAC protocols prevent such collisions.

Figure 2 shows the CDF of RTT for each provider. Both providers have high delay. For EVDO, the median RTT is 150 ms. For WiMax, it is roughly 40 ms. Even this lower of the two delays is surprising given that the path end points are in the same city. We find using *traceroute* that nearly all of this delay is inside the wireless carriers' networks; in fact, a significant fraction is to the first IP-level hop from the wireless client. Details of this experiment are in our extended report [27]. This high delay has implications for how losses can be masked.

We also see that the two links have disparate loss and delay characteristics. This disparity creates significant complications if we want to use them simultaneously. For instance, the factor of three difference in the path RTTs implies that a scheme like round robin will perform poorly. It will significantly reorder packets and unnecessarily delay some packets even though a shorter path exists. Sending all data on the shorter path may not be possible due to capacity constraints, and as is the case in our setup, the lower delay path may have more loss. Using multiple links from the same provider can alleviate the disparity in path properties, but it also reduces reliability because of correlated losses [34].

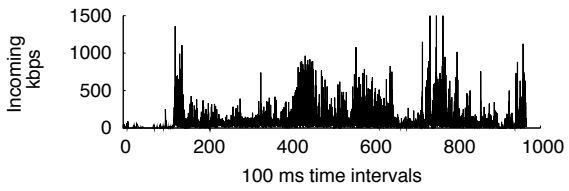


Figure 3: Traffic from Internet to clients

2.2 Workload characteristics

To understand the workload in our target environment, we collect traffic logs from two corporate commuter buses. These buses are different from the ones in our testbed. They have the setup shown in Figure 1, with one Sprint-based 1xEVDO NIC. We sniffed the intra-bus WiFi network for two weeks to capture packets that are sent and received by the riders.

We find that this workload is dominated by short TCP flows [27] which are highly vulnerable to packet loss. It is also highly bursty, as illustrated by an example 100-second period in Figure 3. The average load over the entire trace is quite low, only 86 Kbps, which implies that there is ample leftover capacity along these paths on average. However, short-term load is often above 1 Mbps, which indicates short-term capacity limitations given the throughput that EVDO can achieve. This burstiness means that short-term spare capacity is bursty as well and can differ substantially from the long-term average.

2.3 Discussion

In summary, we characterize our network environment as follows: (i) paths are lossy; (ii) paths have high delays such that timely feedback on packet loss is not available; (iii) the workload is highly bursty such that while there is plenty of capacity available on average, the utilization can approach 100% at short time scales; and (iv) if multiple paths are used, different paths may have different loss and delay properties.

Improving application performance in this environment requires that we reduce packet loss experienced by applications. We could urge the wireless carriers to further improve the lower-layer reliability mechanisms and handoff protocols. This is a long-term proposition that requires significant investment and does not help today's users. We thus build a high performance system on top of existing unreliable connectivity. Improvements to lower-layer connectivity are complementary to our approach.

3. Limitations of existing options

There has been much work on improving application performance over lossy paths. The set of proposed schemes can be broadly classified as those that use retransmissions and those that use erasure coding.

3.1 Retransmission based methods

One way to combat packet loss is by having the sender retransmit lost packets based on feedback from the receiver. This method is used, for instance, in TCP and its variants.[1] However, retransmission based recovery is too slow in settings with high delays. Loss recovery takes at least 1.5 times the round trip time (RTT). We show later that this delay leads to poor performance.

Some methods reduce this delay by isolating the lossy segment of the path such that retransmissions can be performed more quickly. Such retransmissions can be done using support from the wireless base stations (e.g., Snoop TCP [1], Flow Aggregator [7], Ack Regulator [8]) or using additional proxies (e.g., Split TCP [21]). We cannot use these techniques because we do not have access to the wireless carrier's infrastructure. As long as we are sitting outside this infrastructure, the lossy segment of the path will have high delay as well. Thus, the performance of techniques like Split TCP is similar to using an end-to-end TCP connection. We have verified this behavior via experiments in our setting.

Tsao and Sivakumar propose to retransmit lost TCP segments on one interface via another [42]. Their proposal does not use coding is limited to mobile phones, requiring significant changes to TCP stacks on both ends.

3.2 Erasure coding methods

In environments with high delay, erasure coding is a better fit [2]. Erasure coded packets are sent proactively to guard against losses. However, existing erasure coding methods are capacity-oblivious. Systems such as Maelstrom [2] or CORE [23] transmit a fixed number of coded packets for a given set of data packets. If their coding overhead is too low, they do not provide sufficient protection even though there may be excess capacity in the system. If it is too high, they hurt goodput by stealing capacity from data packets.

Even adaptive systems such as MPLOT [38] or LT-TCP [41] adapt to path loss rate and not to spare capacity. Based on the expected loss rate, they add enough redundancy such that data is delivered with a high probability. But because losses as well as spare capacity are bursty, at any given time these systems too can provide insufficient protection even though spare capacity exists or hurt goodput when there is capacity pressure (§6.2).

We argue that the most effective way to protect against losses is to use *all* spare capacity [26]. However, the bursty nature of traffic and thus of spare capacity implies that it is not sufficient to match the level of redundancy to average spare capacity. The short-term mismatch can be significant, leading either to insufficient protection or to

[1] Some experimental TCP variants do not reduce sending rate for non-congestion losses, but their reliability mechanism is still based on retransmissions.

overload. Hence, we develop opportunistic erasure coding (OEC) that provides as much protection as the available capacity allows at short time scales without hurting data packets.

Rateless erasure codes such as LT [24] can generate an unlimited stream of coded packets, but they are not complete erasure coding systems. One must still decide when and how many coded packets to transmit. We also point out later why these codes are inappropriate if one wanted to opportunistically use leftover spare capacity.

4. Opportunistic erasure coding

OEC is meant for lossy environments in which timely feedback on which packets were lost is not available to the sender. Our current design assumes that all packets are equally important; extending OEC to unequal protection (e.g., for video codecs) is a subject of future work.

An ideal goal for an erasure coding scheme in a setting with short TCP flows is to minimize connection completion time, as that directly impact user experience. However, no practical method can achieve this goal when traffic, losses, and path capacity are highly dynamic. We thus modulate our goal to be greedy goodput maximization: each transmission should maximize the amount of new data at the receiver. We show later that this strategy leads to significant reduction in connection completion time.

OEC requires an estimate of the usable capacity of the path. This capacity is not necessarily the physical capacity of the path but is what the OEC traffic can use along the path without hurting others. It may be either be configured or estimated. In PluriBus, we estimate it using a technique based on recent bandwidth measurements tools (§5.2). It can also be estimated using other techniques, e.g., those similar to TCP Vegas [4].

We first describe how OEC functions in the case of one path between the sender and receiver and then describe the generalization to multiple paths.

4.1 Single path case

Consider the following idealized protocol, of which OEC is a practical instantiation. This protocol views network path as a communication channel whose bottleneck capacity matches the given usable capacity. It transmits new data packets as soon as they are generated by the application. If new data is being generated at a rate faster than the channel capacity, it will be queued at the bottleneck. The protocol transmits an erasure coded packet as soon as and only if the packet will find an empty queue. In this way, it uses for coded packets any and all leftover capacity at short time scales. Finally, it encodes each coded packet in a way that maximizes the amount of new data recovered at the receiver.

We argue that this protocol greedily maximizes goodput. By using all capacity, it achieves the highest possible throughput (i.e., rate of unique + non-unique data). Whether it maximizes goodput depends thus on the order and contents of the packets sent. In terms of order, strictly prioritizing data packets, as we do above, is optimal. The reception of a data packet provides one new data packet to the receiver and of a coded packet provides less than one on average [24]. Some coded packets may yield more than one but the average yield will be less than one. Finally, each coded packet is constructed to maximize the amount of new data recovered by the receiver. Thus, in combination, no other protocol can achieve higher goodput at each step, without future knowledge.

To implement this protocol, we need two capabilities. First, we need a method to estimate when the bottleneck queue, which is not necessarily local, will be empty. The knowledge of path capacity and past data and coded transmissions lets us estimate the number of OEC packets at the bottleneck at any given time. We can then transmit coded packets such that they reach the bottleneck when there are no other packets. This way, coded packets always defer to data packets and delay them by at most one packet, while providing as much protection as the amount of spare capacity allows.

In the extreme case where data packets are generated at a rate faster than capacity for an extended period, OEC sends no coded packets. This behavior is optimal with respect to our goal of greedy goodput maximization. However, a certain fraction of coded packets can be easily added to the stream if some minimum protection against loss is desirable at all times. Note, however, that if the end hosts are using end-to-end congestion control, the data rate is unlikely to stay higher than capacity for an extended period if the loss rate experienced is high.

The second capability is an encoding technique that maximizes the amount of data with each coded packet. Conventional erasure codes, whether rateless (e.g., LT [24]) or not (e.g., Reed-Solomon [33]), cannot be used for this purpose. These codes are designed for efficient recovery. They seek to minimize the number of packets needed at the receiver to recover all data. But they recover very little if fewer than the needed threshold number of packets are received [36]. In our setting, with bursty data arrivals, we cannot even predict how many coded packets can be transmitted, let alone how many can be received.

We develop an encoding that greedily maximizes the expected amount of new data recovered by each coded packet. It does that by explicitly accounting for what information might already be present at the receiver. Conventional codes do not consider receiver state at intermediate points in time. We describe our encoding next.

4.2 Greedy encoding

Consider a point in time when the sender has sent a window W of data packets and some coded packets con-

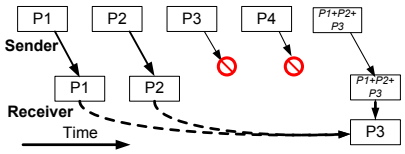


Figure 4: Illustration of our coding system. Data packets P3 and P4 are dropped in transit. The receiver is able to recover P3 after it receives coded packet P1+P2+P3.

structed per our scheme. The sender has not received any feedback from the receiver about the packets in W, and so it is unaware of the exact fate of each data packet. (In §5.1, we describe how W is updated as the sender sends data and coded packets and receives feedback from the receiver.) The receiver has a given data packet either if it received the original transmission of the data packet or if it recovered the packet using a subsequent coded transmission after the original transmission was lost. An example is shown in Figure 4.

Now, the sender has an opportunity to send one more coded packet. Our aim is to construct a coded packet that is "most useful" to the receiver. To keep encoding and decoding operations simple, like several other erasure codes (e.g., LT [24], Growth [19], Maelstrom [2]), we construct coded packets by XOR-ing data packets. Further, to keep analysis simple, we assume that the receiver discards coded packets that cannot be immediately decoded using the data packets that it has received or recovered in the past. The implementation can buffer such packets and decode them later, but we found that this optimization brings little additional gain in our environment.

Thus, to construct a coded packet, the sender must decide which data packets to XOR such that the resulting coded packet is likely to yield a previously missing data packet when decoded using data packets already at the receiver. From a sender's viewpoint, the optimal solution to this problem depends on the probability of each data packet being available at the receiver. This probability is in general different for different packets. It depends on the path loss process, and the precise sequence of coded packets transmitted thus far. It is computationally hard for the sender i) to track these probabilities, as the number of possible combinations grows exponentially; and ii) optimally encode based on individual probabilities.

For tractability, the sender makes a simplifying assumption that each data packet has the same probability, r, of being present at the receiver. We explored heuristics that account for different per-packet probabilities, but found that the performance hit of this assumption is negligible for loss rates and encoding windows sizes that occur in practice. In §5.1, we describe how the sender can estimate r based on path loss rate and past transmissions.

With this assumption, the problem of determining the composition of an ideal coded packet boils downs to how many packets should be XOR'd [10]. Suppose the sender XORs c data packets. The probability that this coded packet will yield a previously missing data packet at the receiver equals the probability that exactly one out of the c packets is missing. Thus, the expected yield of this coded packet is:

$$Y(c) = c \cdot (1-r) \cdot r^{c-1} \quad (1)$$

To maximize the expected yield, we have:

$$c = -1/ln(r)$$

This result can be intuitively explained. Observe that c is inversely proportional to r. If the fraction of data packets at the receiver is low, we construct a coded packet by XOR-ing few data packets. For instance, if most packets are missing, the best strategy is to encode only one packet (i.e., send a duplicate); coding even two is sub-optimal as the chance of both being absent and nothing being recovered is high. Conversely, if a higher fraction of packets are present at the receiver, encoding more packets recovers missing data faster.

Thus, the sender randomly selects $\max(1, \lfloor \frac{-1}{ln(r)} \rfloor)$ data packets to XOR. We round down because including fewer data packets is safer than including more.

4.3 Generalizing to multiple paths

OEC can be generalized to the case where transmissions are spread over multiple paths with disparate loss and delay characteristics, while maintaining the greedy goodput maximization property.

In the presence of multiple paths, we send each data packet along the path that currently offers the least delay [9], which is judged using estimates of queue length and propagation delay. We continue to send traffic along the fastest path until queuing increases its delay to the level of the next fastest path, and so on. This method naturally generalizes striping mechanisms such as round robin to the case of paths with different delays and capacities. It minimizes average packet delay, and makes reordering less likely. Variations and mis-estimations of path delay can still lead to some reordering, which we handle using a small sequencing buffer. Coded packets are sent as before, when spare transmission opportunities open up along any path.

We argue that the use of delay-based striping in OEC greedily maximizes goodput. Let there be k paths between the two proxies and let the capacity, delay, and loss rate of path i be c_i, d_i and p_i respectively. The least delay selection policy then creates a virtual path whose capacity is equal to the sum of the individual capacities, delay is less than the maximum individual delay, and the loss rate is the weighted average of individual loss rates [9]. That is, $C = \sum_{i=1}^{k} c_i$, $D \leq \max_{i=1}^{k} d_i$, and $P \leq \sum_{i=1}^{k} \frac{p_i * c_i}{\sum_{i=1}^{k} c_i}$. This combination is optimal with

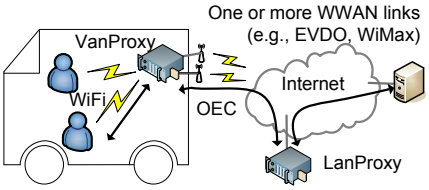


Figure 5: The architecture of PluriBus. It uses OEC between the two proxies and can combine multiple WWAN links for additional capacity.

respect to goodput [9]. OEC on top of this virtual path greedily maximizes goodput, as it does for a single path.

4.4 Applying OEC

Applying OEC in an environment requires three tasks: i) specify W and r for greedy encoding; ii) estimate the bottleneck queue length to guide when coded packets are transmitted; and iii) if multiple paths exist, estimate current delay along each, so that the least delay path is used for each data packet. We describe below how we conduct these tasks in the vehicular environment, which is particularly challenging because it is highly dynamic.

OEC is designed to use all spare capacity to improve user performance. As such, it is more appropriate for settings where i) the underlying transmission channel isolates users from one another, as is the case for WWAN MACs; ii) the incremental cost of sending data is small, as is the case with fixed-price, unlimited-usage data plans. We revisit this issue in § 5.5.

5. PluriBus: OEC in moving vehicles

Figure 5 shows the architecture of PluriBus. The VanProxy is equipped with one or more WWAN links. All packets are relayed through LanProxy, which is located on the wired Internet.[2] Such relaying allows us to mask packet losses on the wireless links without modifying the remote computers to run PluriBus. OEC is used between the two proxies for data flowing in both directions.

We describe below how we accomplish the three tasks for applying OEC. Our methods for the latter two tasks borrow heavily from prior work.

5.1 Specifying W and r for greedy encoding

The sender initializes W=ϕ (i.e., empty set) and r=0 and updates these values when a data or coded packet is sent or feedback is received from the receiver.

[2]Relaying via LanProxy may increase end-to-end latency. However, because of the high delay inside wireless carrier networks, any increase is small if the LanProxy is deployed in the same city. Internet paths within a city tend to be short [40]. Interestingly, relaying through our deployed LanProxy actually reduced latency to most destinations due to Detour effects [37].

i) When a new data packet is sent, it is inserted in W and then r is updated to reflect the probability that the new packet is received. More precisely:

$$r \leftarrow ((|W|-1)\cdot r + (1-p))/|W|$$

where p is a rough estimate of the loss rate of the path along which the packet is sent. Receivers estimate p using an exponential average of past behavior and periodically inform the sender of the current estimate. Burstiness of losses can complicate the task of estimating loss rates. Our experiments show that PluriBus is robust to the inaccuracies that we find in practice [27].

ii) When a coded packet, formed by XOR-ing c data packets, is sent, W does not change, and r is updated to reflect the probability that the coded packet is received, and yielded a new packet. That is:

$$r \leftarrow (|W|\cdot r + (1-p)\cdot Y(c))/(|W|)$$

where $Y(c)$ is the expected yield of the packet (Eq. 1).

iii) When the receiver returns the highest sequence number that it has received, which is embedded in packets flowing in the other direction (§5.4), packets with lower or equal sequence numbers are purged from W. We reset r to p.

The purge from W ensures that the sender encodes only over data packets generated roughly in the last round trip time. Because higher-layer protocols such as TCP detect losses and initiate recovery at this time-scale, it avoids duplicate recovery of packets. Thus, even though OEC logically uses all spare capacity, in practice it may not. No coded packets are sent when W is empty, that is, no new data packets have arrived in the last RTT.

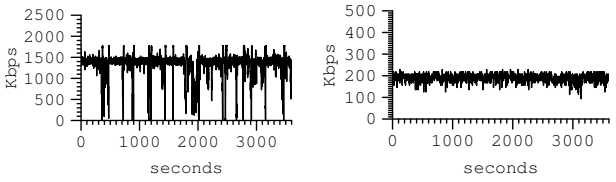
5.2 Estimating queue length

We maintain an estimate of queue length along a path in terms of the *time* required for the bottleneck queue to fully drain our packets. It is zero initially and is updated after packet transmissions:

$$Q \leftarrow \frac{(PacketSize * PathCapacity)}{max(0, Q - TimeSinceLastUpdate)}$$

$PathCapacity$ refers to the capacity of the path. The capacity of a path is the rate at which packets drain from queue at the bottleneck link. It is different from throughput, which refers to the rate at which packets reach the receiver. The two are equal only in the absence of losses. We conservatively estimate path capacity using a simple method described below.

The WWAN MAC protocols control media usage by individual transmitters, making it easier to estimate capacity than CSMA-based links (e.g., WiFi). As an example, Figure 6 shows the throughput of WiMax paths in the two directions for a one-hour window in which we generate traffic at 2 Mbps in each direction. We see roughly stable peak throughputs of 1500 and 200 Kbps, which



(a) Downlink (b) Uplink

Figure 6: WiMax downlink and uplink throughputs. The y-axis ranges of the two graphs are different.

correspond to their capacity. Incoming sequence numbers confirm that throughput dips are due to packet losses and not slowdowns in queue drain rate. For a detailed analysis of 3G link capacity, see [22].

Our capacity estimation is conservative, so that PluriBus is more likely send fewer coded packets than sending too many. The estimator, like other bandwidth estimation tools [16, 17], is based on a simple observation: if the sender sends a train of packets faster than the path capacity, the receive rate corresponds to the path capacity. However, unlike prior tools [16, 17], we do not use separate probe traffic. Instead, we rely on the burstiness of data traffic and the capacity-filling nature of OEC to create packet trains with a rate higher than path capacity.

We bootstrap the proxies with expected path capacities. The receiver measures the rate of incoming packets and computes the sending rate using the transmission timestamp in each packet. The two rates are computed over a fixed time interval (500 ms). The capacity estimate is updated based on intervals in which the sending rate is higher than the current estimate. If the receive rate is higher than the current estimate for three consecutive intervals, the estimate is increased to the median rate in those three intervals. Similarly, if the receive rate is lower for three consecutive intervals, the estimate is decreased to the median rate. Because our sending rate equals at least our estimated capacity, when actual capacity is lower, the estimate is downgraded quickly. Changes in capacity estimate are communicated to the sender.

Errors in capacity estimate can lead to errors in the queue length estimate. In theory, this error can grow unboundedly. In practice, we are aided by periods where little or no data is transmitted, which are common with current workloads. Such periods reset the estimate to its correct value of zero. While we cannot directly measure the accuracy of our queue length estimate, we show in §6.3.2 that our path delay estimate, which is based on it, is fairly accurate.

5.3 Identifying minimum delay path

When spreading data across multiple paths, PluriBus needs to estimate the current delay along each path. A simple method is to use the running average of one-way delays observed by recent packets, based on feedback from the receiver. However, we find that this method is quite inaccurate (§6.3.2) because of feedback delay and because it cannot capture with precision short time scale processes such as queue build-up along the path. Capturing such processes is important to consistently select the path with the minimum delay.

Our estimate of path delay is based on i) transmission time, which primarily depends on path capacity; ii) queue length; and iii) propagation delay. We described above how we estimate the first two. Measuring propagation delay requires finely synchronized clocks at the two ends, which may not be always available. We skirt this difficulty by observing that we can identify the faster path even if we only compute the propagation delay plus a constant that is unknown but same across all paths. This constant happens to be the current clock skew between the two proxies.

Let the propagation delay of a path be d and the (unknown) skew between the two proxy clocks be δ. We estimate $d + \delta$ based on Paxson's method [30]. A packet that is sent by the sender at local time s will be received by the receiver at local time r, where $r = s + d + \delta + Q + \frac{PacketSize}{PathCapacity}$. If there is no queuing, $d + \delta = r - s - \frac{PacketSize}{PathCapacity}$. We can thus compute $d + \delta$ using local timestamps of packets that see an empty queue.

To enable the estimate above, the receivers keep a running exponential average of $r - s - \frac{PacketSize}{PathCapacity}$ (i.e., $d + \delta$) for each path. Only packets that have likely sampled an empty queue are used for computing the average. Packets that get queued at bottleneck link arrive roughly $\frac{PacketSize}{PathCapacity}$ time units after the previous packet. We use in our estimates packets that arrive at least twice that much time after the previous packet. The running average is periodically reported by the receiver to the sender.

It is now straightforward for the sender to compute the path with least delay. This path is the one with the minimum value of $\frac{PacketSize}{PathCapacity} + Q + (d + \delta)$, which is in fact an estimate of the reception time at the receiver.

5.4 Implementation

We now briefly describe our implementation of PluriBus. We encountered and overcame many interesting engineering challenges while deploying PluriBus on our testbed. For instance, we need to correctly handle frequent IP address changes for the VanProxy in a way that is transparent to users and maintains their connections across changes. Due to lack of space, we omit most of these details from this paper and document them separately [27].

The VanProxy and the LanProxy create a bridge between them, and tunnel packets over the WWAN paths between them. The IP packets sent by users and remote computers are encapsulated within UDP packets that are sent over these paths. We do not use lower-overhead IP-in-IP tunneling as our wireless carriers block them. The

UDP packets include a special header that contains timestamps and additional information to allow the other end to correctly decode and order received packets. Each proxy caches incoming and decoded data packets for a brief period (50ms). This cache allows it to decode coded packets and temporarily store out of order packets. In-order data packets are relayed immediately. Those received out of order are relayed as soon as the previous packet is relayed or upon expiration from the cache.

The PluriBus header and the encapsulation lowers the effective link MTU by 41 bytes, which may lead to fragmentation issues (similar to those with VPNs). To minimize fragmentation, we inform the clients of the lower MTU via DHCP. Some clients inform their wide area peers of their MTU during TCP connection establishment, via the MSS option. For other clients, we are experimenting with modifying the MSS option of TCP SYNs as they traverse the VanProxy. With these changes, only large UDP packets destined for the clients, which constitute a small fraction in our traces, will be fragmented.

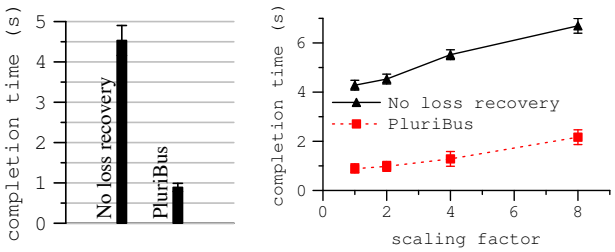
5.5 Discussion

PluriBus aggressively uses spare capacity. If transit operators subscribe to fixed-price, unlimited-usage plans, this "selfish" design maximizes user performance. However, if they have a usage-based plan, their costs will increase. In theory, this increase can be significant because OEC logically fills the pipe. But in practice PluriBus is not constantly transmitting because it encodes only over data in the last round trip time. We show later that PluriBus increases usage by only a factor of 2 for realistic workloads, with the increment being lower when the baseline demand is higher. We expect that transit operators would be willing to pay extra for better performance, as the cost of wireless access is likely a small fraction of their operational cost and amortizes over many users.

6. Evaluation

We now evaluate PluriBus. We show that it significantly improves application performance (§6.1) and that OEC outperforms loss recovery based on retransmissions or capacity-oblivious erasure coding (§6.2). We also provide microbenchmarks for some aspects of PluriBus (§6.3).

Experimental platforms: We deployed PluriBus on two buses that operate regularly on a corporate campus (§2). Each bus has one WiMax link and one EVDO link. The observed average loss rate is 5% for WiMax and under 1% for EVDO, though it can be bursty. The round trip delays are 40 and 150 ms respectively. The variations in path loss, delay and capacity are all natural; we do not control them in any way. A computer placed on each bus generates the workload described below. Because of support and manageability issues, we were not allowed to carry real user traffic on our experimental system. These two buses are our primary platform for studying the performance of PluriBus in a real environment. For more extensive experimentation and to consider different environments, we complement it with controlled experiments using a network emulator. To avoid confusion, we label our results with "Deployment" or "Emulator," depending on the platform used for the experiment.



(a) Measured workload (b) Scaled workload

Figure 7: Benefit of loss recovery in PluriBus. [Deployment]

Workload: For the experiments in this paper, we generate realistic, synthetic workloads from the traces described in §2.2. We first process the traces to obtain distributions of connection sizes and inter-arrival times, where a connection is the standard 5-tuple. The synthetic workload is based on these distributions of connection sizes and inter-arrival times [12]. The average demand of this workload is 86 Kbps but it is highly bursty.

To verify if our conclusions apply broadly, we also experimented with other workloads. These include workloads with a fixed number of TCP connections and those generated by a synthetic Web traffic generator [3]. The results are qualitatively similar to those below.

Performance measure: We use connection completion time as the primary measure of performance. It is of direct interest to interactive traffic such as short Web transfers that dominate the vehicular environment.

This paper uses the mean to aggregate performance across trials and connections. To show that the differences in means are statistically significant, we plot confidence intervals as well. Results that plot median and interquartile ranges can be found in our extended report [27].

6.1 Benefit of PluriBus

We start by studying the benefit of PluriBus compared to the current practice of not using any loss recovery (beyond end-to-end TCP). We study other loss recovery mechanisms in the next section. The results in this section are based on our deployment.

Figure 7(a) shows connection completion times for PluriBus and without any loss recovery. The latter uses delay-based path selection [9].[3] These results are based

[3]Today, more capacity, if needed, is added by installing addi-

on over four weeks of data. Each method operated for at least four days and completed tens of thousands of connections.

The graph shows the mean and 95% confidence intervals (CI) around the mean computed using Student's t distribution. We see that PluriBus significantly reduces completion time. Its mean completion time is under 1 second compared to over 4 seconds without loss recovery. This reduction represents a relative improvement factor of over 4 and an absolute improvement of over 3 seconds.

The reduction in completion time due to PluriBus can significantly improve user experience. Web transactions tend to have multiple connections (some sequential, some parallel) and even tens of milliseconds of additional delay can impact users' interaction with some Web sites [20].

Though not shown here, we find that PluriBus reduces the loss rate seen by end hosts to almost zero (0.3%). Without loss recovery, this loss rate is over 3%.

Benefit under higher load: Since the gains of PluriBus stem from using spare path capacity, an interesting question is whether these gains disappear as soon as the workload increases. To study the performance of PluriBus as a function of load, we scale the workload by scaling the inter-arrival times. To scale by a factor of two, we draw inter-arrival times from a distribution in which all inter-arrival times are half of the original values, while retaining the same connection size distribution. Our workload synthesis method does not capture many details, but it captures the primary characteristics that are relevant for our evaluation. We find that the performance for a synthetic workload scaled by a factor of 1 is similar to an exact replay of connection size and arrival times.

Figure 7(b) plots the mean and 95% confidence intervals of flow completion time as a function of the scaling factor used for the workload. Across both buses, these results are based on over four weeks of data. Each data point is based on at least two days. We see PluriBus performs well even when the workload is scaled by a factor of eight. In fact, its performance at that extreme level is better than what the absence of loss recovery offers without scaling the workload at all. Even at such high load levels, there is ample instantaneous spare capacity for PluriBus to mask losses by sending coded packets and improve performance (see §6.3.1). The loss rate seen by end hosts is roughly 0.5% with PluriBus, while it is 3% without any loss recovery.

tional VanProxies, each with its own WWAN link. Each user connects to exactly one VanProxy (i.e. an AP) and all her traffic is exchanged through that VanProxy. This policy balances load poorly because it operates at the granularity of users and often there are only a handful of active users. Our experiments (not shown here) confirm that its performance is poorer than that of delay-based path selection.

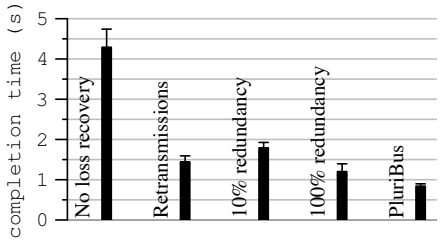


Figure 8: Performance of various loss recovery mechanisms. The graph plots the mean and (the top end of) 95% confidence interval for completion time. [Emulation]

6.2 Other loss recovery mechanisms

Having seen that loss recovery brings significant benefits in the vehicular environment, we now compare the use of OEC in PluriBus to other loss recovery mechanisms. We consider both retransmission based and erasure coding based loss recovery.

In retransmission-based loss recovery, the receiving proxy reports to the sender which packets have not been received, at which point the sender retransmits them. Both original packets as well as retransmissions are sent along the path that we currently estimate as offering the least delay. This policy provides an upper bound on policies such as pTCP [15] that do retransmission-based loss recovery because it uses the least delay channel and does not reduce the sending rate in response to losses.

The second loss recovery method that we consider is the capacity-oblivious erasure coding. We implement a code with $K\%$ redundancy by sending a coded packet after every $\frac{100}{K}$-th pure packet. Each coded packet codes over packets in the current unacknowledged window since the last coded packet. Thus, when K=10, every 11^{th} packet is coded. This code is identical to $(K, 1)$ Maelstrom code [2]. Both coded and pure packets are sent over the path with the least estimated delay.

The experiments in this section are based on emulation of the characteristics of wireless paths that we observe in our deployment. As described earlier (§2, §5.2), we have collected detailed traces to study the loss rate, delay and capacity of the wireless links in our testbed. We drive the emulation by updating emulated link's delay, loss, and capacity every second, as observed in the traces. The workload is as before, based on our traces.

Figure 8 shows the results. Notice that the "No Loss Recovery" and "PluriBus" bars are similar to those from deployment experiment (Figure 7(a)), which suggests that our emulation methodology is able to recreate the essential characteristics for these links.

We see that OEC-based loss recovery in PluriBus outperforms loss recovery based on both retransmissions and capacity-oblivious erasure coding. Compared to retransmissions, OEC's mean completion time is lower by 0.6 seconds (reduction factor of 1.7) because its loss recov-

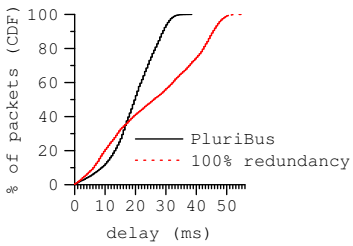


Figure 9: Delay experienced by data for two loss recovery methods. The graph plots the observed delay minus the minimum observed. [Emulation]

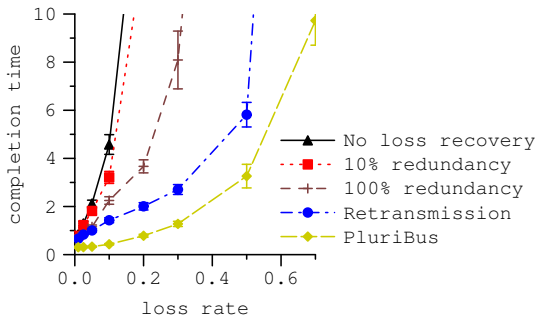


Figure 10: Performance of different loss recovery methods as a function of loss rate. The graph plots the mean and 95% confidence interval for completion time. [Emulation]

ery is faster. Compared to using erasure coding with 10% redundancy, its mean completion time is lower by 0.95 seconds (reduction factor of 2). This level of redundancy does poorly because it does not recover from many losses. Even though the average loss rate is low, the loss process is bursty and in periods of higher loss rates, using 10% redundancy is not sufficient. Data show that the post-recovery loss rate is 1.5%.

Compared to erasure coding with 100% redundancy, the mean completion time of PluriBus is lower by 0.4 seconds (reduction factor of 1.4). This level of redundancy is able to recover from most losses in our environment. Data show that the post-recovery loss rate is 0.5%. But by not being opportunistic, it imposes a higher queuing delay on data packets. This effect is shown in Figure 9, which plots the one-way delay, in addition to the minimum observed, for the two methods. We see that the 100% redundancy imposes a much higher delay on data.

The poorer performance of both ends of the redundancy levels relative to PluriBus, for different reasons, underscores the challenge in extracting good performance with capacity-oblivious erasure coding.

Impact of path loss rate: The results above demonstrate that OEC outperforms other loss recovery schemes under realistic path conditions. We now evaluate if the performance advantage of OEC persists in a range of settings with different loss rates.

To isolate the impact of loss rate, we perform emulation experiments with a single link between the two proxies. The link has a one-way delay of 75 ms and capacity of 1.5 Mbps. The loss rate on the link is varied from 1% to 70%. We show results using the Gilbert-Elliot (GE) loss model that induces bursty losses. Simpler loss models in which each packet has the same loss probability yield qualitatively similar results [27]. The GE model has two states, a good state with no (or low) loss and a bad state with high loss. The model is specified using the loss rate in the two states and state transition probabilities. We set both the transition probabilities to 0.5 and vary the loss rate of the bad state.

Figure 10 shows the results as a function of loss rate. We see that PluriBus outperforms other loss recovery methods across the board. These results also show that OEC performs better than capacity-oblivious erasure coding even if the coding overhead of these methods is adapted to loss rate. If we were to tune the overhead to expected loss rate, the overhead of the two erasure coding methods that we study must be suitable for some loss rate, but we see that OEC is better in the entire range. The reason is as explained earlier. Consider, for example, adding 100% redundancy. When less than half of the channel capacity is being used by data packets, OEC adds more than 100% redundancy and thus provides better protection, especially to the loss of many packets in a short time window. When more than half the channel is being used, OEC adds less than 100% redundancy. For the same amount of (coded+data) traffic that successfully reaches the receiver, the OEC traffic has more data than 100% redundancy traffic. The combined effect is that OEC tends to perform better than any capacity-oblivious redundancy method across a broad range of loss rates.

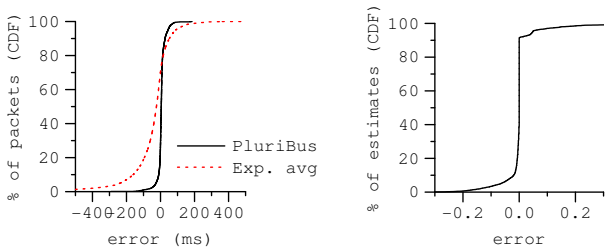
6.3 Understanding PluriBus in detail

We have studied the behavior of PluriBus in detail. In this paper, we report on the extra data sent by PluriBus due to coding and the accuracy of our delay and loss estimators. We defer to [27] other investigations such as the impact of inaccuracies in loss and delay estimates on performance, specific coding and decoding strategies we use, and fine-tuning of their parameters.

6.3.1 Amount of coded packets transmitted

Given that PluriBus is logically capacity filling, how many coded packets does it actually generate? Using data from the experiment in Figure 7(b), we find that the average percentage of coded packets is 54%. At scaling factors of 1, 2, 4 and 8, the percentage of coded packets is 67, 60, 57 and 35. Thus, as expected, PluriBus reduces the fraction of coded packets as workload increases because there are fewer opportunities to send coded packets.

We also find that while PluriBus logically fills the pipe,



(a) Path delay (b) Path loss rate

Figure 11: Error in estimating path delay and loss in PluriBus. [Deployment]

the actual amount of coded traffic is much lower because it codes over only data packets that arrive in the last RTT. At the scaling factor of 1, the average packet transmission rate of PluriBus is 258 Kbps, which is much lower than the combined capacity of the two links.

6.3.2 *Accuracy of path delay estimation*

Various factors, including estimates of path capacity, queue length, and propagation delay, impact the delay estimate of PluriBus. For good performance, the accuracy of this estimate is important. We evaluate accuracy by comparing the estimated delivery time at the sender to the actual delivery time at the receiver. This comparison is possible even with asynchronous clocks because our estimate of propagation delay includes the clock skew.

Figure 11(a) shows delay estimation error (i.e., estimate minus actual) in our deployment. It includes all load scaling factors; results are similar across all factors. The curve labeled PluriBus shows that our estimate is highly accurate, with 80% of the packets arriving within 10 ms of the predicted time. This is encouraging, especially considering the inherent variability in the delay of WWAN paths [22]. As a result of this accuracy, we find that fewer than 5% of the packets arrive out of order at the receiver and 95% of the out-of-order packet have to wait less than 10 ms in the sequencing buffer.

The curve marked "Exp. avg." shows the error if delays were estimated simply as an exponential average of observed delays, rather than our more precise accounting based on estimated capacity and queue length. Note that it tends to significantly underestimate path delay. We find that this underestimation significantly degrades performance, to a level that is sometimes worse than not using any loss recovery.

6.3.3 *Accuracy of loss rate estimation*

PluriBus uses an estimate of loss rate to estimate r, the probability of a packet being at the receiver, which is used in greedy encoding. Given the dynamics of the vehicular environment, loss rate maybe hard to estimate. Figure 11(b) shows that we obtain accurate estimates of loss rate in our deployment. It plots the difference in the loss rate for the next twenty packets minus the current running average of the loss rate that we use to predict future loss rate. Over 90% of the time, our estimate is within ± 0.1.

7. Additional related work

We now outline work that we build on, in addition to the work on combating path losses that we summarized earlier (§3).

Inverse multiplexing: Like PluriBus, many systems combine multiple links or paths into a single, high-performance communication channel. PluriBus differs primarily in its context and the generality of the problem tackled—our paths have disparate delays, capacities, and loss rates. Most existing works assume identical links [11], identical delays [39], or ignore losses [9, 34, 31].

A few systems, such as pTCP, R-MTP or MTCP, stripe data between end hosts across arbitrary paths by using TCP or a similar protocol along each path [15, 25, 5, 32]. Loss recovery is done via retransmissions. As we showed in §6, because of high path delays, this approach performs worse than PluriBus.

Delay-based striping, which we use to generalize OEC to multiple paths, was proposed by Chebrolu and Rao [9]. We combine it with loss recovery, which we find is important for it to be effective.

Improving connectivity for vehicles: Like us, MAR [34] and Horde [31] combine multiple WWAN links to improve vehicular Internet access. MAR showed the value of using multiple links using simple connection-level striping. It left open the task of building higher-performance algorithms. PluriBus employs one such algorithm (OEC). Horde [31] specifies a QoS API and stripes data as per policy. It requires that applications be re-written to use the API, while we support existing applications. Neither MAR nor Horde focus on loss recovery.

Some researchers have focused on improving WLAN (WiFi) connectivity to moving vehicles using lower-layer techniques such as rate adaptation and directional antennae [6, 29]. In contrast, we focus on WWAN links and on improving connectivity for applications by masking the deficiencies of connectivity provided by lower layers.

Erasure code: Numerous erasure codes have been proposed in the literature. The encoding used in OEC is a generalization of Growth codes [19] that were designed to transmit data in large sensor networks with failing sensors. Our generalizations include an explicit consideration of loss rate and data already at the receiver. The optimal degree of a coded packet (§4.2) is also derived by Considine [10]. However, that work does not address any of the systems issues (e.g., when to transmit packets).

8. Conclusions

Opportunistic erasure coding (OEC) is a new erasure coding scheme that varies the amount of coding overhead to fit the instantaneous spare capacity along a path. We built and deployed PluriBus, which applies OEC to a vehicular context, and found that it reduces the mean flow completion time by a factor of 4 for realistic workloads.

While we focused on the vehicular context, OEC is a general technique that can be used in other lossy environments where timely feedback is not available, e.g., wireless multicast and satellite links. Further, the two core mechanisms in OEC—opportunistic transmissions and greedy encoding—may be independently useful. Opportunistic transmissions can be used to transfer other kinds of low-priority data such that it uses only the capacity leftover by high-priority data. Greedy encoding can be used in other dynamic environments (e.g., wireless meshes) where the number of packets that will be received cannot be predicted in advance. We plan to study these possibilities in the future.

9. References

[1] H. Balakrishnan et al. Improving TCP/IP performance over wireless networks. In *MobiCom*, 1995.

[2] M. Balakrishnan et al. Maelstrom: Transparent error correction for lambda networks. In *NSDI*, 2008.

[3] P. Barford and M. Crovella. Generating representative Web workloads for network and server performance evaluation. In *SIGMETRICS*, 1998.

[4] L. S. Brakmo and L. L. Peterson. TCP Vegas: End to end congestion avoidance on a global Internet. *IEEE JSAC*, 13(8), 1995.

[5] K. Brown and S. Singh. M-TCP: TCP for Mobile Cellular Networks . *CCR*, 27(5), 1997.

[6] J. Camp and E. Knightly. Modulation rate adaptation in urban and vehicular environments: cross-layer implementation and experimental evaluation. In *MobiCom*, 2008.

[7] R. Chakravorty, S. Katti, J. Crowcroft, and I. Pratt. Using TCP flow aggregation to enhance data experience of cellular wireless users. *IEEE JSAC*, 23(6), 2005.

[8] M. C. Chan and R. Ramjee. TCP/IP performance over 3G wireless links with rate and delay variation. *Wireless Networks (Kluwer)*, 11(1-2), 2005.

[9] K. Chebrolu and R. Rao. Bandwidth aggregation for real-time applications in heterogeneous wireless networks. *IEEE ToMC*, 4(5), 2006.

[10] J. Considine. Generating good degree distributions for sparse parity check codes using oracles, 2001.

[11] J. Duncanson. Inverse multiplexing. *IEEE Comm. Mag.*, 32(4), 1994.

[12] J. Eriksson et al. Feasibility study of mesh networks for all-wireless offices. In *MobiSys*, 2006.

[13] CIRA 3G mobile broadband router. http://www.feeneywireless.com/products/routers/cira/cira.php.

[14] N. Graychase. Greyhound launches in-bus Wi-Fi. http://www.wi-fiplanet.com/news/article.php/3736816.

[15] H.-Y. Hsieh and R. Sivakumar. ptcp: An end-to-end transport layer protocol for striped connections. In *ICNP*, 2002.

[16] N. Hu et al. Locating Internet bottlenecks: Algorithms, measurements, and implications. In *SIGCOMM*, 2004.

[17] M. Jain and C. Dovrolis. End-to-end available bandwidth: measurement methodology, dynamics, and relation with TCP throughput. In *SIGCOMM*, 2002.

[18] K. Jang et al. 3G and 3.5G wireless network performance measured from moving cars and high-speed trains. In *MICNET*, 2009.

[19] A. Kamra et al. Growth Codes: Maximizing Sensor Network Data Persistence. In *SIGCOMM*, 2006.

[20] R. Kohavi et al. Practical guide to controlled experiments on the web: Listen to your customers not to the hippo. In *SIGKDD*, 2007.

[21] S. Krishnamurthy, M. Faoutsos, and S. Tripathi. Split TCP for Mobile AdHoc Networks. In *Globecom*, 2002.

[22] X. Li et al. Experiences in a 3G network: interplay between the wireless channel and applications. In *MobiCom*, 2008.

[23] Y. Liao, Z. Zhang, B. Ryu, and L. Gao. Cooperative robust forwarding scheme in DTNs using erasure coding. In *MILCOM*, 2007.

[24] M. Luby. LT Codes. In *FOCS*, March 2002.

[25] L. Magalhaes and R. Kravets. Transport level mechanisms for bandwidth aggregation on mobile hosts. In *ICNP*, 2001.

[26] R. Mahajan et al. Eat all you can in an all-you-can-eat buffet: A case for aggressive resource usage. In *HotNets*, 2008.

[27] R. Mahajan et al. E pluribus unum: High performance connectivity on buses – extended version. MSR-TR-2009-76, June 2009.

[28] Does wi-fi on transit attract riders? http://www.masstransitmag.com/interactive/2010/11/04/does-wi-fi-on-transit-attract-riders/.

[29] V. Navda et al. Mobisteer: using steerable beam directional antenna for vehicular network access. In *MobiSys*, 2007.

[30] V. Paxson. *Measurements and Analysis of End-to-end Internet dynamics*. PhD thesis, UC Berkeley, 1997.

[31] A. Qureshi and J. Guttag. Horde: separating network striping policy from mechanism. In *MobiSys*, 2005.

[32] C. Raiciu, M. Handley, and D. Wischik. Coupled multipath-aware congestion control. IETF draft draft-raiciu-mptcp-congestion-01.txt, Mar. 2010.

[33] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *SIAM*, 8(2), June 1960.

[34] P. Rodriguez et al. MAR: A commuter router infrastructure for the mobile Internet. In *MobiSys*, 2004.

[35] F. Salmon. Did wifi cause a rise in bus ridership? http://blogs.reuters.com/felix-salmon/2011/12/26/did-wifi-cause-a-rise-in-bus-ridership/, 2011.

[36] S. Sanghavi. Intermediate performance of rateless codes. In *Information Theory Workshop*, 2007.

[37] S. Savage et al. The end-to-end effects of Internet path selection. In *SIGCOMM*, 1999.

[38] V. Sharma et al. MPLOT: A transport protocol exploiting multipath diversity using erasure codes. In *INFOCOM*, Apr. 2008.

[39] A. Snoeren. Adaptive inverse multiplexing for wide-area wireless networks. In *Globecom*, 1999.

[40] N. Spring, R. Mahajan, and T. Anderson. Quantifying the causes of path inflation. In *SIGCOMM*, 2003.

[41] V. Subramanian et al. An end-to-end transport protocol for extreme wireless network environments. In *MILCOM*, 2006.

[42] C. Tsao and R. Sivakumar. On effectively exploiting multiple wireless interfaces in mobile hosts. In *CONEXT*, 2009.

[43] WiFi incompatibility on Seattle metro? http://www.internettablettalk.com/forums/archive/index.php?t-18539.html.

[44] Sound transit riders. A mailing list for Microsoft employees.

[45] Google's buses help its workers beat the rush. http://www.nytimes.com/2007/03/10/technology/10google.html.

[46] Metro bus riders test county's first rolling WiFi hotspot. http://www.govtech.com/e-government/King-County-Metro-Bus-Riders-Test.html.

[47] Microsoft WiFi-enabled system will debut this month. http://seattlepi.nwsource.com/business/330745_msfttranspo07.html.

[48] Soundtransit – Wi-Fi. http://www.soundtransit.org/Rider-Guide/Wi-Fi.xml.

Server-assisted Latency Management for Wide-area Distributed Systems

Wonho Kim[1], KyoungSoo Park[2], and Vivek S. Pai[1]

[1]*Department of Computer Science, Princeton University*
[2]*Department of Electrical Engineering, KAIST*

Abstract

Recently many Internet services employ wide-area platforms to improve the end-user experience in the WAN. To maintain close control over their remote nodes, the wide-area systems require low-latency dissemination of new updates for system configurations, customer requirements, and task lists at runtime. However, we observe that existing data transfer systems focus on resource efficiency for open client populations, rather than focusing on completion latency for a known set of nodes. In examining this problem, we find that optimizing for latency produces strategies radically different from existing systems, and can dramatically reduce latency across a wide range of scenarios.

This paper presents a latency-sensitive file transfer system, Lsync that can be used as synchronization building block for wide-area systems where latency matters. Lsync performs novel node selection, scheduling, and adaptive policy switching that dynamically chooses the best strategy using information available at runtime. Our evaluation results from a PlanetLab deployment show that Lsync outperforms a wide variety of data transfer systems and achieves significantly higher synchronization ratio even under frequent file updates.

1 Introduction

Low-latency data dissemination is essential for coordinating remote nodes in wide-area distributed systems. The systems need to disseminate new data to remote nodes with minimal latency when distributing new configurations, when coordinating task lists among multiple endpoints, or when optimizing system performance under dynamically changing network conditions. All of the scenarios involve *latency-sensitive synchronization*, where the enforced synchronization barrier can limit overall system performance and responsiveness.

The effects of synchronization latency will become increasingly more important as more Internet services leverage distributed platforms to accelerate their applications in the WAN. Recent trends show that many companies employ distributed caching services [4, 18] and WAN optimization appliances [24], or deploy their applications at the Internet edges close to the end users [12] to improve the user experience. These platforms should handle frequently-changing customer requirements and adapt to dynamic network conditions at runtime. For instance, Akamai [2] reports that its management server receives five configuration updates per minute that need to be propagated to remote CDN nodes immediately [26]. If these systems face long synchronization delays, possibilities include service disruption, inconsistent behavior at different replicas visible to end users, or increased application complexity to try to mask such effects.

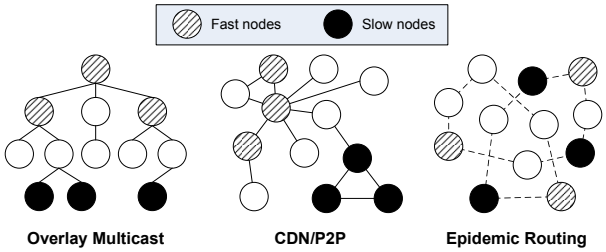


Figure 1: Slow Nodes in Overlay – Peering strategies in scalable one-to-many data transfer systems are not favorable to slow nodes.

The latency is measured as the total completion time of file transfer to all target remote nodes. In wide-area systems, it is usual that a number of nodes experience network performance problems and/or lag far behind up-to-date synchronization state at any given time. We observe these *slow nodes* typically dominate the completion time, which means that managing their tail latency is crucial for latency-sensitive synchronization.

Although numerous systems have been proposed for scalable one-to-many data transfers [8, 10, 16, 17, 19, 21], they largely ignore the latency issue because re-

source efficiency is typically their primary concern for serving an open client population. In an *open client population*, there is no upper bound on the number of clients, so the systems aim to maximize average performance or aggregate throughput in the system.

As a result, those systems are not favorable to slow nodes (Figure 1). For instance, many overlay multicast systems attempt to place well-provisioned *fast nodes* close to the root of the multicast tree while pushing slow nodes down the tree. Likewise, the peering strategies used in CDN/P2P systems make slow nodes have little chance to download from fast nodes. The random gossiping in epidemic routing protocol helps slow nodes peer with fast nodes, but only for a short-term period. These peering strategies not only produce long completion time but make the systems highly vulnerable to changes in slow node's network condition. Despite these disadvantages, it is common that existing services rely on one of the data transfer schemes for coordinating their remote nodes.

In this paper, we explore general file transfer policies for latency-sensitive synchronization with the goal of minimizing completion time in a *closed client population*. This completion time metric drives us to examine new optimization opportunities that may not be advisable for systems with open client populations. In particular, we aggressively use spare bandwidth in the origin server to assist nodes that experience transient/persistent performance problems in the overlay mesh at runtime. The server allocates its bandwidth in a manner favorable to the slow nodes while synchronizing the other nodes through existing overlay mesh. This server-assisted synchronization reduces the tail latency in slow nodes without sacrificing scalable data transfers in the overlay mesh, which drastically improves the completion time and achieves stable file transfer.

For evaluation of our policies, we develop Lsync, a low-latency file transfer system for wide-area distributed systems. Lsync can be used as synchronization building block for wide-area distributed systems where latency matters. Lsync continuously disseminates files in the background, monitoring file changes and choosing the best strategy based on information available at runtime. Lsync is designed to be easily pluggable into existing systems. Users can specify a local directory to be synchronized across remote machines, and give Lsync the information about target remote nodes. Other systems can use Lsync by simply dropping files into the directory monitored by Lsync when the files need latency-sensitive synchronization.

We evaluate Lsync against a wide variety of data transfer systems, including a commercial CDN. Our evaluation results from a PlanetLab [1] deployment show that Lsync can drastically reduce latency compared to existing file transfer systems, often needing only a few seconds for synchronizing hundreds of nodes. When we generate a 1-hour workload similar to the frequent configuration updates in Akamai CDN, Lsync achieves significantly higher synchronization ratio than the alternatives throughout the experiment.

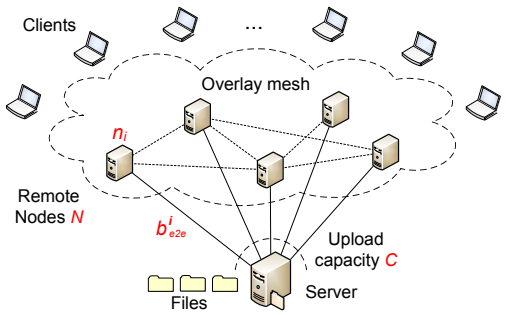


Figure 2: Synchronization Environment – the server has files to transfer to remote nodes with low completion latency. The remote nodes construct an overlay mesh for providing services to external clients.

The rest of this paper is organized into following sections. In Section 2, we describe the model and assumptions for our problem. Based on the model, we discuss how to allocate server's bandwidth to slow nodes in Section 3. In Section 4, we describe how to divide nodes between server and overlay mesh in a way to minimize the total latency. We describe the implementation in Section 5 and evaluate it on PlanetLab in Section 6.

2 Synchronization Environment

In this section, we describe the basic operational model used for Lsync, along with the assumptions we make about its usage.

Operational model We assume one dedicated server that coordinates a set of remote nodes, N, geographically distributed in the WAN, as shown in Figure 2. The management server has an uplink capacity of C, and can communicate with each remote node n_i with bandwidth b^i_{e2e}. The C value is configurable, and represents the maximum bandwidth that can be used for remote synchronization. The b^i_{e2e} values are updated using a history-based adaptation technique, described in Section 4.5, to adjust to variations in available bandwidth. The server knows the amount of new data by detecting file changes in the background as described in Section 5.

Many distributed systems construct an overlay mesh among their remote nodes to use scalable data transfer systems. If an overlay mesh is available and certain conditions are met, Lsync leverages it for fast synchronization. To make Lsync easily pluggable into existing systems, we do not require modification of a given overlay's behavior. Instead, Lsync characterizes the overlay

mesh using a black-box approach to estimate its startup latency. Based on the estimation, Lsync determines how to adjust workload across the server and the overlay.

Target environments This model is appropriate for our target environments, where a large number of nodes are geographically distributed without heavy concentrations in any single datacenter.[1] Our target environments also require that we exclude IP multicast, due to the problems stemming from lack of widespread deployment and coordination across different ASes.

Examples of current systems where our intended approach may be applicable include distributed caching services [4, 18], WAN optimization appliances [24], distributed computation systems [25], edge computing platforms [12], wide-area testbeds (PlanetLab, OneLab), and managed P2P systems [20]. In addition, with most major switch/router vendors announcing support for programmable blades for their next-generation networks, virtually every large network will soon be capable of being a distributed service platform.

Target remote nodes In Lsync, users can specify target remote nodes in various ways. For global updates, users will configure Lsync to optimize the latency for updating all remote nodes. For partial updates, it is possible to select a subset of specific remote nodes to synchronize. Users can also specify a certain fraction of nodes that need to be synchronized fast for satisfying certain consistency models or incremental rollouts of new configurations in the system. We name the parameter *target synchronization ratio* denoted by r, for the rest of the paper. This enables Lsync to use intelligent node selection, which is particularly effective for disseminating frequent updates.

3 Server Bandwidth Allocation

Lsync's file transfer policy combines multiple factors that contribute to the overall time reduction. We quantify the benefits separately, so we describe steps separately in the paper. After adjusting workload across server and overlay, Lsync exploits the server's spare bandwidth to speed up synchronizing the overlay mesh. In this section, we begin by examining the effects of the server's bandwidth allocation policies on completion latency.

Figure 3 shows an example of the timeline when the server transfers files to a set of target remote nodes, $N_{target} \subset N$. The server detects a new file f_{new} at time t_0. Each horizontal bar corresponds to a remote node, $n_i \in N_{target}$, that has variable-sized unsynchronized data f^i_{prev} remaining from previous transfers. The areas of f^i_{prev} and f_{new} represent the sizes of the files, $|f^i_{prev}|$ and $|f_{new}|$, respectively. The height of the bar, b^i_{e2e}, is the end-to-end bandwidth from the server to n_i that starts its transfer after a pending time t^i_{pend}. At any given time t, the sum of the heights of the bars should not exceed the server's upload capacity C in order to avoid self-congestion and associated problems stemming from the server overload [3].

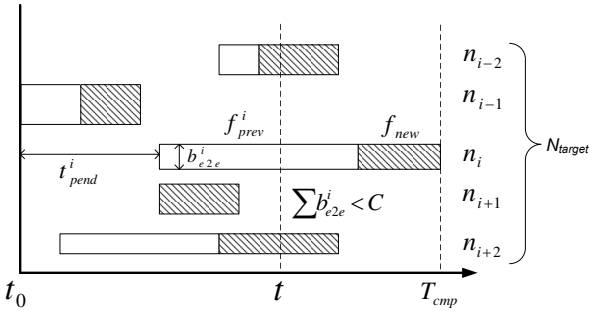


Figure 3: Timeline of Synchronization – The completion time T_{cmp} is determined by the node with the latest finish time among the target nodes N_{target}.

The completion time, T_{cmp}, is calculated as

$$T_{cmp} = t_0 + \max_i \left(t^i_{pend} + \frac{|f^i_{prev}| + |f_{new}|}{b^i_{e2e}} \right) \quad (1)$$

for $n_i \in N_{target}$. There are two variables that the server can control transparently to the remote nodes. The server can determine t^i_{pend} that n_i should wait before starting its transfer. The server can also select nodes for N_{target} if a target synchronization ratio is given. From the perspective of the server, controlling these two variables corresponds to *node scheduling* and *node selection* policies in the server, respectively. The basic intuition behind the policies is that we could reduce the latency by giving low t^i_{pend} to slow nodes and carefully selecting nodes for N_{target}. In the following sections, we compare different policies using real measurements on PlanetLab to quantify their effects on latency.

3.1 Node Scheduling

We begin by examining how we schedule transfers when the number of transfers exceeds the outbound capacity of the server. This is the problem of minimizing makespan, which is NP-hard [7]. We compare two basic scheduling heuristics, Fast First and Slow First. The *Fast First* is similar to Shortest Remaining Processing Time (SRPT) scheduling in that the server's resource is allocated to the node who has the shortest expected completion time. SRPT is known to be optimal for minimizing mean response time [6]. The *Slow First* is the opposite of the Fast First policy, and selects the nodes with the longest expected completion time when the server has available bandwidth.

[1] The intra-datacenter bandwidths are sufficient to make the synchronization latency less of an issue in that environment.

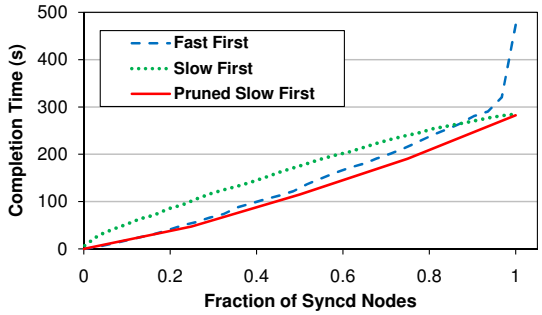


Figure 4: Node Scheduling and Node Selection – Pruned Slow First captures both the initial speed advantage of Fast First, as well as the total overall advantage of Slow First.

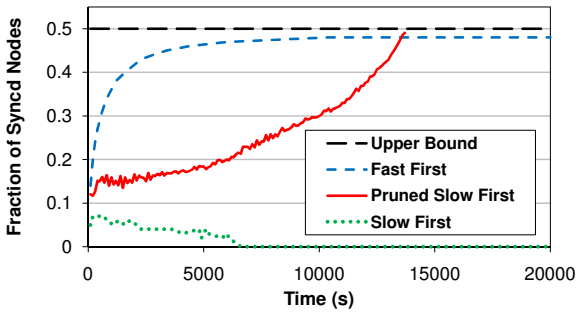


Figure 5: Synchronizing Frequent Updates – While Fast First synchronizes quickly at first, Pruned Slow First actually reaches the upper bound more quickly.

We compare the two schedulers on PlanetLab nodes. We implement the policies in a dedicated server that has 100 Mbps upload capacity. Then we generate two files – f_{new} (1 MB), and f_{prev} (10 MB) that has been in the process of synchronization on the nodes, from 1% to 99% complete. We measure the time to synchronize all live PlanetLab nodes (559 nodes at the time of the experiments) with either of the two scheduling modes enabled.

The result of the measurement is shown in Figure 4. Fast First synchronizes most nodes faster than Slow First, but at high target ratios, Fast First performs much worse. The reason for the difference is somewhat obvious: near the end of the transfers in Fast First, only slow nodes remain, and the server's uplink becomes underutilized. This underutilization occurs for the final 175 seconds in Fast First, but only for 0.5 seconds in Slow First. We also evaluated *Random* scheduling, which selects a random node to allocate available bandwidth. From 10 repeated experiments, we found that Random scheduling yields completion times between Fast First and Slow First, but generally closer to Slow First for all target ratios.

This result implies that slow nodes dominate the completion time in the WAN and that the Slow First scheduling can mask their effects on latency. Another implication of the result is that offline optimization will provide little benefit in the scenario. Note that the server's bandwidth is underutilized only for 0.5 seconds in Slow First. This means that no scheduler can reduce the latency by more than 0.5 seconds in the setting. In addition, our evaluation results show that runtime adaptation has a significant impact on latency, which offline schedulers cannot provide.

3.2 Node Selection

In this section, we examine the effect of node selection. In Lsync, users can specify their requirements in the form of target synchronization ratio, a fraction of nodes that need to be synchronized fast. If the ratio is given, Lsync attempts to find a subset of nodes that could further reduce latency.

We show that integrating node selection with Slow First scheduling can blend the best behaviors of Slow First and Fast First. Given target ratio r, we first sort all nodes in an increasing order of the estimated remaining synchronization time. We then pick $N_s \cdot r$ nodes where N_s denotes the number of nodes in N, and use Slow First scheduling for the selected nodes. The remaining $N_s \cdot (1-r)$ nodes are synchronized using Slow First as well after the selected nodes are finished. As a result, the completion time does not suffer either from the slow synchronization in the beginning (Slow First) or the long tail at the end (Fast First). We name the integrated scheme *Pruned Slow First* for comparison. Figure 4 shows that Pruned Slow First outperforms the other scheduling policies across all target ratios.

We examine a dynamic file update scenario where new files are frequently added to the server, which is common in large-scale distributed services. For an in-depth analysis, we use simulations for the experiment. We generate 2000 nodes with bandwidths drawn from the distribution of the inter-node bandwidths on PlanetLab.

We study how these frequent updates affect the synchronization process. Given information about available resources, we can determine how much change the server can afford to propagate. We define *update rate*, u, as the amount of new content to be synchronized per unit time (one second). Then, $N_s \cdot u$ is the minimum bandwidth required to synchronize all N_s nodes with u rate of change. The upper bound on the achievable synchronization ratio is $\frac{C}{N_s \cdot u}$ where C is the server's upload capacity. C is set to 100 Mbps in the experiment.

Figure 5 shows each policy's performance under frequent updates. As before, the server has two files, 1 MB and 10 MB that are in the process of synchronization, and new files are constantly added to the server with an update rate 100 Kbps. The upper bound is 0.5 in this setting, and no policy can reach beyond this limit.

We see that the completion time drastically changes as we use different policies. In particular, the synchronization ratio of Slow First drops over time and reaches

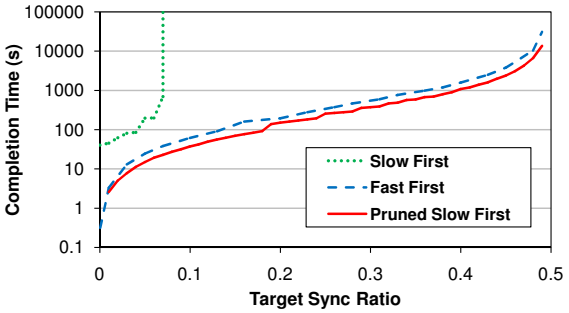


Figure 6: Synchronization Latency for Frequent Updates – While Slow First leads to failure, integrating node selection with the Slow First scheduling reduces latency for all target ratios (y-axis is in log-scale).

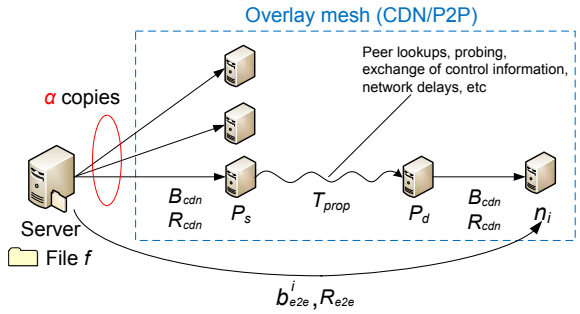


Figure 7: Startup Latency in CDN/P2P – To leverage a given overlay system, Lsync estimates the startup latency for fetching a new file f from the server and propagating to the remote nodes n_i in the overlay.

0 near 6800 seconds because it gives high priority to the nodes that are far behind up-to-date synchronization state. Fast First synchronizes nodes quickly in the beginning, but asymptotically approaches the upper bound since the remaining slower nodes make little forward progress given the rate of change.

To examine the performance of Pruned Slow First, we set r to 0.49, which is slightly below the upper bound 0.5 in this setting. In Figure 5, Pruned Slow First is worse than Fast First in the beginning, but it reaches its target ratio much earlier than the other policies. Pruned Slow First reduces the completion time by 56% compared with Fast First, showing that the best policy can provide significant latency gains, while the worst policy, Slow First, actually leads to failure in this scenario. In Figure 6, we show the completion time of each policy for a range of feasible r values. The Pruned Slow First outperforms the other policies for every target ratio (the y-axis is in log-scale).

We showed that the Slow First scheduling helps reduce completion latency, and integrating node selection with the Slow First scheduling can further reduce latency particularly when handling frequent updates. Based on the observations, we extend our discussion to examine how to leverage overlay mesh for latency-sensitive synchronization in the next section.

4 Leveraging Overlay Mesh

With a better understanding of the bandwidth allocation policies in the server, we focus on understanding how to leverage scalable one-to-many data transfer systems which we collectively name CDN/P2P systems for the rest of the paper. Many large-scale distributed services construct an overlay mesh for scalable data transfer to external clients, and often use the overlay mesh for internal data dissemination to remote nodes as well [26]. In this section, we explore how to leverage the overlay mesh to reduce synchronization latency without changing its behaviors.

4.1 Startup Latency in Overlay Mesh

To leverage a given overlay mesh, Lsync needs to predict *startup latency* for distributing a new file that is not cached on the remote nodes in the overlay. However, it is difficult to predict the accurate latency since CDN/P2P systems typically have diverse peering strategies and dynamic routing mechanisms. To allow easy integration with existing systems, Lsync uses a black-box approach to characterizing a given overlay mesh to estimate its startup latency.

Figure 7 presents a general model showing how a new file f in the server is propagated to a remote node n_i via overlay mesh. P_s is a peer node contacting the server to fetch f, and P_d is a peer node that n_i contacts. If f is already cached in P_d, n_i will receive it directly from P_d. However, in latency-sensitive synchronization, all target nodes attempt to fetch f as soon as it is available in the server. In case f should be fetched from the server, the CDN/P2P system selects node P_s to contact the server. Depending on the system's configuration, multiple copies of the file can be fetched into the overlay mesh. The fetched file is propagated from P_s to P_d possibly through some intermediate overlay nodes, and delivered to n_i. T_{prop} is the propagation delay from P_s to P_d. In addition to the network delay between peer nodes, T_{prop} also includes other overheads such as peer lookups and exchange of control messages, which are system-specific. The bandwidth and RTT between CDN/P2P peer nodes are B_{cdn} and R_{cdn} respectively.

For the simplicity of the model, we begin with an ideal assumption that nodes in CDN/P2P are uniformly distributed, and thus the bandwidth and RTT between neighboring nodes are constants, B_{cdn} and R_{cdn}. However, we will adjust the parameter values later to account for their variations on real deployment in the WAN. This model is not tied to a particular CDN/P2P system or any specific algorithms such as peer selection and request redirection. We apply the model to different types of deployed CDN/P2P systems in Section 6.2. We show

that the model captures the salient characteristics of these systems, and that Lsync can adjust its transfers to utilize each of these systems.

4.2 Completion Time Estimation

To schedule workload across server and overlay mesh, Lsync first estimates the expected completion time in the overlay mesh. The *setup cost*, δ, measures the first-byte latency for fetching newly-created content through the overlay mesh. Specifically, δ is defined as

$$\delta = 2 \cdot R_{cdn} + T_{prop} \tag{2}$$

The overall overlay completion time, T_{cdn}, can be calculated as follows: To distribute f to n_i, the server informs n_i of the new content availability, taking time R_{e2e}. Then, n_i contacts P_d, and starts receiving the file with delay δ. Delivering the entire file to n_i with bandwidth B_{cdn} requires time $\frac{f_s}{B_{cdn}}$ where f_s denotes the size of f. The total time, T_{cdn} is then the sum,

$$T_{cdn} = R_{e2e} + \delta + \frac{f_s}{B_{cdn}} \tag{3}$$

Note that T_{cdn} does not depend on r because all target nodes fetch f simultaneously via the overlay. In comparison, the end-to-end completion time, $T_{e2e}(r)$, for transferring f to remote nodes using Pruned Slow First is

$$T_{e2e}(r) = R_{e2e} + \frac{N_s \cdot r \cdot f_s}{C} \tag{4}$$

where C is the server's upload capacity.

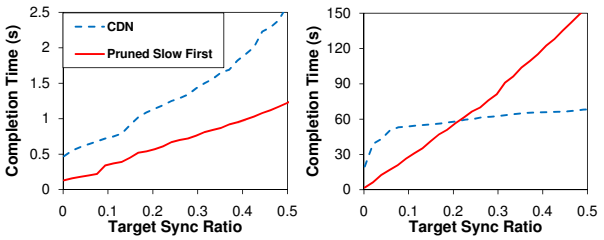
4.3 Selective Use of Overlay Mesh

If a given CDN/P2P system has high startup latency, it will outperform end-to-end transfers only when its bandwidth efficiency can outweigh the cost. After the server estimates T_{cdn} and $T_{e2e}(r)$, the server can dynamically choose between end-to-end transfers and the overlay mesh to get better latency. In this section, we examine the conditions that such a selective use of overlay mesh can provide benefits in a real deployment.

To get a more accurate estimation of the completion time, we extend T_{cdn} in (3) to reflect the fact that the bandwidth distribution of a CDN/P2P system is not uniform. Rather than modeling each node's bandwidth separately, we use the minimum bandwidth value for the top $N_s \cdot r$ nodes, yielding

$$T_{cdn}(r) = R_{e2e} + \delta^r + \frac{f_s}{B^r_{cdn}} \tag{5}$$

B^r_{cdn} is the $N_s \cdot r$ th largest B_{cdn}, and δ^r is the $N_s \cdot r$ th smallest setup cost. As we increase r, by definition, B^r_{cdn} will monotonically decrease, and δ^r will monotonically increase.



(a) Small file (5 KB) (b) Large file (5 MB)

Figure 8: End-to-End Connections vs. Overlay Mesh – For small file, the latency of overlay mesh is hampered by the long setup time, but its efficient bandwidth usage outweighs the cost for large file.

Now we can compare $T_{cdn}(r)$ with $T_{e2e}(r)$ and then use either end-to-end connections or the overlay mesh as appropriate. The decision process can be formally defined as following. The server uses end-to-end connections when either of the following conditions is met.

$$f_s < \delta^r \cdot \frac{C \cdot B^r_{cdn}}{N_s \cdot r \cdot B^r_{cdn} - C} \tag{6}$$

$$B^r_{cdn} < \frac{C}{N_s \cdot r} \tag{7}$$

From the above test conditions, we can draw the following general guidelines. Using end-to-end connections is better when (1) the file size f_s is small, (2) the overlay setup cost δ^r is large, (3) the server's upload capacity C is high, (4) the target synchronization ratio r is low, (5) the client population N_s is small, or (6) the overlay bandwidth B^r_{cdn} is significantly smaller than the server's bandwidth.

We examine the potential benefits of the scheme by comparing end-to-end transfers with CoBlitz CDN [19] on PlanetLab. Beyond PlanetLab, CoBlitz has been used in a number of commercial trial services [11], and we believe CoBlitz represents one of the typical CDN services currently available. For end-to-end transfers, we use Pruned Slow First policy for allocating the server's bandwidth.

Figure 8 shows the latency for synchronizing a small file (5 KB) and a large file (5 MB). For a small file, using end-to-end transfers shows better performance than the CDN because the completion time of the CDN is hampered by its setup cost. When transferring a large file, the CDN shows better performance since its efficient bandwidth usage outweighs the setup cost in the CDN.

Wide-area systems typically disseminate files of varying sizes. For instance, Akamai management server has file transfers spanning 1 KB to 100 MB. The measurement results in Figure 8 imply that Lsync will need different strategies for different file sizes to address the tradeoffs between startup latency and bandwidth efficiency of the overlay mesh.

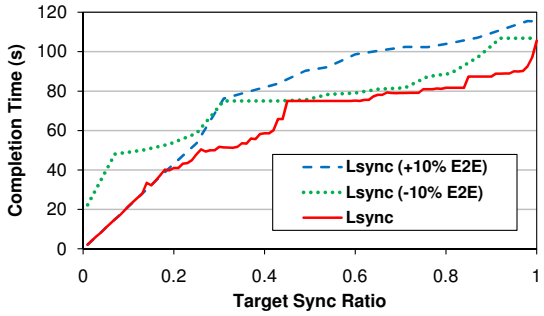


Figure 9: Optimality of the Division – The split between overlay and E2E is not improved by moving some nodes to the other mechanism, suggesting that Lsync's split is close to optimal.

4.4 Using Spare Bandwidth in Server

When nodes are being served by the overlay mesh, the origin server load is greatly reduced, leading to spare bandwidth that can then be used to serve some nodes bypassing the overlay. Given r, Lsync divides remote nodes into two groups. One group directly contacts the origin server, and the other group downloads the file via the overlay mesh. Lsync adds nodes with the worst overlay performance to the end-to-end group until the expected end-to-end completion time matches the overlay completion time.

More formally, Lsync calculates r_{e2e}, the ratio of nodes to place in the end-to-end group. After r_{e2e} is determined, Lsync selects the $N_s \cdot (r - r_{e2e})$ nodes having the fastest overlay connections. From (5), the overlay completion time is estimated as $T_{cdn}(r - r_{e2e})$. The remaining $N_s \cdot r_{e2e}$ nodes will use end-to-end connections. To estimate the spare bandwidth in the server, we consider *CDN load factor* α, which represents how many copies are fetched from the origin server into the overlay mesh. Different systems have different load factors, and the CoBlitz fetches 9 copies from the origin server. As the origin server is supposed to send α copies of f to the overlay group, Lsync should reserve B_{load} bandwidth which is $\frac{\alpha \cdot f_s}{T_{cdn}(r-r_{e2e})}$ for overlay traffic, yielding a value of $(C - B_{load})$ for the server's spare bandwidth. Using this spare bandwidth, the end-to-end completion time is

$$T_{e2e}(r_{e2e}) = R_{e2e} + \frac{N_s \cdot r_{e2e} \cdot f_s}{C - B_{load}} \qquad (8)$$

Then, Lsync calculates r_{e2e} that makes the two groups complete at the same time.

We simulate synchronizing PlanetLab nodes using the bandwidths and setup costs measured in all PlanetLab nodes. Figure 9 shows a sensitivity analysis of r_{e2e}, with two other simulations for slightly higher and lower values of r_{e2e}, in sending a 5 MB file. Lsync outperforms both for all target ratios, suggesting that it is choosing the optimal balance of the two groups.

4.5 Adaptive Switching in Remote Nodes

To mitigate the effect of real-world bandwidth fluctuations, we add a dynamic adaptation technique to Lsync. The main observation behind the technique is that minor variations in performance do not matter for most nodes, since most nodes will not be the bottleneck nodes in the transfer. However, when a node that is close to being the slowest in the overlay group becomes slower, it risks becoming the bottleneck during file transfer. The lower bound on the overlay bandwidth is $B_{cdn}^{r-r_{e2e}}$.

When the origin server informs the remote nodes of new content availability, the server sends $B_{cdn}^{r-r_{e2e}}$ and $T_{cdn}(r - r_{e2e})$. After $T_{cdn}(r - r_{e2e})$ passes, if a node is not finished, it compares the current overlay performance with $B_{cdn}^{r-r_{e2e}}$. If the current performance is significantly lower than the expected lower bound, the node stops downloading from the overlay mesh, and directly goes to the origin server to download the remaining data of the file. In our evaluation, we configure the node to switch to the origin server when its overlay performance drops below 75% of its expected value. Our evaluation results show that using adaptive switching improves the completion time while lowering variations.

5 Implementation

Lsync is a daemon that performs two functions – in one setting, it operates in the background on the server to detect file changes, manage histories, and plan the synchronization process. In its second setting, it runs on all remote nodes to coordinate the synchronization process. Both modes of operation are implemented in the same binary, which is created from 6,586 lines of C code. Lsync daemon implements all the techniques described in the previous sections.

Since Lsync is intended to be easily deployable, it operates entirely in user space. Using Linux's inotify mechanism, the daemon specifies files and directories that are to be watched for changes – any change results in an event being generated, which eliminates the need to constantly poll all files for changes. When the Lsync daemon starts, it performs a per-chunk checksum of all files in all of the directories it has been told to watch using Rabin's fingerprinting algorithm [23] and SHA-1 hashes. Once Lsync is told a file has been changed, it recomputes the checksums to determine what parts of the file have been changed. If new files are created, Lsync receives a notification that the directory itself has changed, and the directory is searched to see if files have been created or deleted.

Once Lsync detects changes, it writes the changes into a log file, along with other identifying information. This log file is sent to the chosen remote Lsync daemons, either by direct transfer, or by copying the log file to a Web-

Systems	δ^r	B^r_{cdn}	Division Ratio	
			E2E	Overlay
CoBlitz	1.4	1.2	0.24	0.76
Coral	7.6	0.6	0.52	0.48
BitTorrent	29.7	4.7	0.30	0.70

Table 1: Division of Nodes between E2E and overlay mesh. r is 0.5, and file size is 5 MB. We also tested small files (up to 30 KB), but E2E outperformed all these systems. δ^r is in seconds, and B^r_{cdn} is in Mbps.

accessible directory and informing the remote daemons to grab the file using a CDN/P2P system. Once the remote daemon receives a log file, it applies the necessary changes to its local copies of the file.

6 Evaluation

In this section, we evaluate Lsync and its underlying policies on PlanetLab. Each experiment is repeated 10 times with each setting, and 95^{th} percentile confidence intervals are provided where appropriate.

6.1 Settings

We deploy Lsync on all live PlanetLab nodes (528 nodes at the time of our experiments), and run a dedicated origin server with 100 Mbps outbound capacity. The server has a 2.4 GHz dual-core Intel processor with 4 MB cache, and runs Apache 2.2.6 on Linux 2.6.23. We measure latency for a set of target synchronization ratios including 0.05, 0.25, 0.5, 0.75, and 0.98. We use a maximum synchronization level of 0.98 to account for nodes that may become unreachable during our experiments. Lsync attempts to achieve the given target ratio fast while leaving other available nodes synchronized via overlay in the background.

6.2 Startup Latency in CDN/P2P Systems

CDN/P2P designers typically expect that the steady state of the CDN/P2P system is that the content is already pulled from the origin, and is being served to clients over a much longer lifetime with high cache hit ratio. However, for synchronization, remote nodes typically request changes that are not in their overlay mesh. Therefore, Lsync monitors the performance of first fetching content from the origin server, which was not a major issue in existing CDN/P2P systems.

Using our black-box model described in Section 4.1, we measured parameters, δ^r and B^r_{cdn}, for two running CDN systems and one P2P system that we deploy on PlanetLab. Table 1 shows the setup costs and bandwidths of the three systems. Each system was characterized by simply fetching new content from the remote nodes, and measuring each node's first-byte latency and bandwidth. The three systems show interesting differences in behavior. BitTorrent spends more time getting the content initially, but then has higher bandwidth. The CDN systems show relatively low setup costs because they were designed for delivering web objects to clients rather than sharing large files.

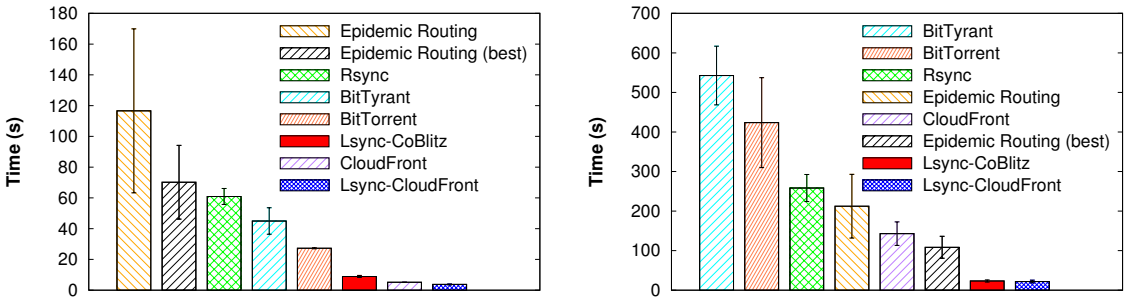
The table also shows how Lsync would allocate nodes between overlay transfers and end-to-end transfers for a synchronization level of 0.5. For small file transfers, Lsync opts to use end-to-end connections rather than any of the systems. For larger transfers, the fraction assigned to direct end-to-end transfers is determined by the tradeoff between latency and bandwidth. For CoBlitz (with the lowest latency) and BitTorrent (with the highest bandwidth), Lsync assigns most nodes to the overlay group. For Coral, with slightly higher latency and lower bandwidth, Lsync assigns nodes roughly equally between overlay and E2E. We select CoBlitz for the rest of our experiments, mostly due to its low latency.

6.3 Comparison with Other Systems

We compare Lsync with different types of data transfer systems. We transfer our CoBlitz web proxy executable file (600 KB) to all PlanetLab nodes using each of the systems, and measure completion times (Figure 10). Each system is designed for robust broadcast (epidemic routing), simple cloning (Rsync), bandwidth efficiency (CDN), and high throughput and fairness (P2P systems). We evaluate the performance of the systems when they are used for latency-sensitive synchronization in the WAN.

Rsync *Rsync* [27] is an end-to-end file synchronization tool widely used for cloning files across remote machines. It uses delta encoding to minimize the amount of transferred data for changes in existing files. In this experiment, however, the server synchronizes a newly generated file not available in remote nodes, so the amount of the transferred data is the same as in the other tested systems. The Rsync server relies only on end-to-end connections to remote nodes for synchronizing the file, and does not use any policies in allocating server's bandwidth. Therefore, the result of Rsync represents the performance of end-to-end transfers with no policy applied.

P2P systems We use *BitTorrent* [9] and *BitTyrant* [21] as examples of P2P systems. Although BitTorrent has a high setup cost (Table 1), it performs better than end-to-end copying tools (Rsync) for target ratio of 0.5 because the majority of nodes have high throughput. However, BitTorrent ends up with very long completion time (424 seconds) and high variations (standard deviation 193) for the target ratio of 0.98. This is because, with BitTorrent, slow nodes have little chance to download from peer nodes with good network conditions, which makes the



(a) Target Sync Ratio 0.50

(b) Target Sync Ratio 0.98

Figure 10: Comparison with Other Systems – We compare Lsync with various data transfer systems in terms of the latency for synchronizing CoBlitz web proxy executable file (600 KB).

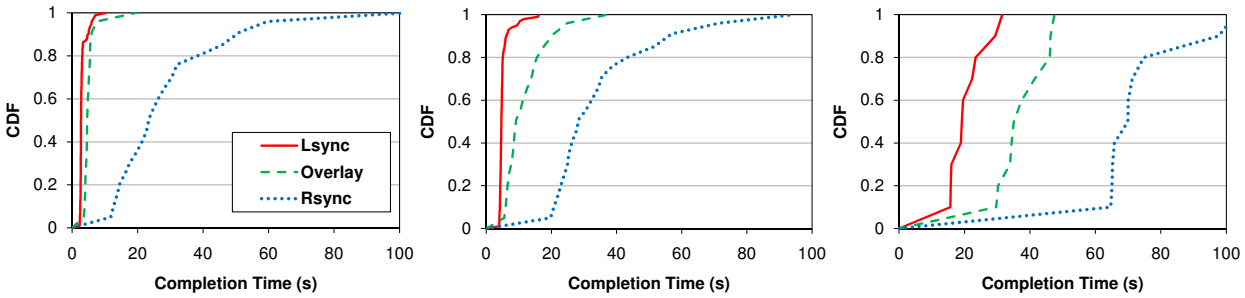
slow nodes finish more slowly than in other systems. We see the similar trend with BitTyrant.

Epidemic routing We also implemented *epidemic routing* (or gossip) protocol [8] to measure its latency in the WAN. In the protocol, each node picks a random subset of remote nodes periodically, and exchanges file chunks. For a conservative evaluation, we assume that every node has the full membership information to avoid membership management, which is known to be the main overhead of the protocol [15]. There are two key parameters in the gossip protocol: how often the nodes perform gossip (*step interval*) and how many peers to gossip with (*fanout*). These two parameters directly impact on the protocol's performance, but it is hard to tune them because of their sensitivity to network conditions. To find the best configuration for our experiments, we tried a range of values for step interval (0.5, 1, 5, and 10 seconds) and fanout (1, 5, 10, 50, and 100 nodes). Then we picked a configuration that generated the shortest latency. Since we use our own implementation of the protocol, we first compared our implementation with published performance results of the protocol in a similar setting. CREW [13] used the gossip protocol for rapid file dissemination in the WAN and outperformed Bullet [17] and SplitStream [10] in terms of completion time in the evaluation. Our implementation showed a comparable latency (141 seconds) to CREW's result (200 seconds) under the same setting (60 nodes, 600 KB file, 200 Kbps bandwidth).[2]

"Epidemic Routing (best)" in Figure 10 represents the results with the best configuration that we found, (1 second interval and 10 fanout), and "Epidemic Routing" shows the average performance of all configurations that we tested (excluding cases with 30 minutes timeout). The result with the tuned configuration always outperforms the average case. Both of the results show interesting patterns in completion time. The protocol works worse than both Rsync and P2P systems at low target ratio because file chunks are disseminated only during gossip rounds. The file data is disseminated relatively slow in the beginning. However, the gossip protocol outperforms the other systems at target ratio of 0.98. Unlike in P2P systems, the slow nodes have better chances to peer with fast nodes during file transfer. As a result, they do not become bottleneck in overall completion time. The random peering policy in the gossip protocol helps slow nodes to catch up with other nodes, but the protocol is typically more optimized for robust dissemination than latency.

Commercial CDNs Since our CDN is deployed on PlanetLab, one may think its performance is adversely affected by some of overloaded PlanetLab nodes. We measure synchronization latency using a commercial CDN, Amazon CloudFront [4]. CloudFront is faster than the other systems for the low target ratio due to its well-provisioned CDN nodes. However, we observe that some nodes are still slow in fetching new file from the CDN. Specifically, 11% of PlanetLab nodes spend more than 20 seconds downloading the file from CloudFront, and the slowest nodes are in Egypt, Tunisia, Argentina, and Australia. As the file is not cached in the CDN, the slow nodes should fetch the file possibly through multiple intermediate nodes in the overlay. This internal behavior depends on system-specific routing mechanisms, which is not visible outside of the overlay. We implemented another version of Lsync, Lsync-CloudFront, which incorporates CloudFront as its underlying overlay mesh. Lsync-CloudFront estimates the overlay's startup latency, and focuses the server's resource on the bottleneck nodes at runtime. We find that the slow nodes can be served better from the server than via the well-provisioned overlay for uncached files, leading to the best performance among the tested systems.

[2] As the topology is not specified, we randomly selected 60 PlanetLab nodes, and averaged over 10 repeated experiments. We used a user-level traffic shaper, Trickle [14], for setting bandwidth on the selected PlanetLab nodes.



(a) 10 KB (b) 100 KB (c) 1 MB

Figure 11: Distribution of Completion Times – For all file sizes, Lsync outperforms the other systems because Lsync adjusts its file transfer policies based on file size as well as network conditions.

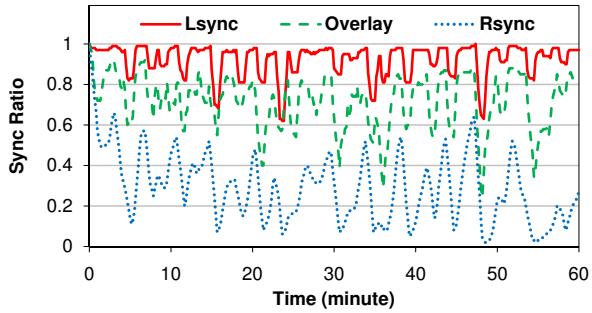


Figure 12: Frequently Added Files – Lsync makes most nodes fully synchronized during the entire period of the experiment.

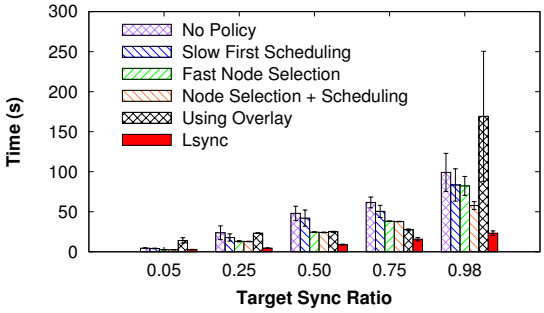


Figure 13: Lsync Contributing Factors – We see that each component in Lsync contributes to the overall time reduction.

6.4 Frequently Added Files

In large-scale distributed services, file updates occur frequently, and new files can be added to the server before the previous updates are fully synchronized across nodes. To evaluate Lsync under frequent updates, we generate a 1-hour workload based on the reported workload of Akamai CDN's configuration updates [26]. A new file is added to the server every 10 seconds, and the file size is drawn from the reported distribution in Akamai. We compare Lsync with CoBlitz and Rsync. The two systems represent alternative approaches: using overlay mesh only (CoBlitz) and using end-to-end transfers only (Rsync).

When a new file is added, we measure how many remote nodes are fully synchronized with all previous files and compute the average of the synchronization ratios over one minute. Figure 12 shows the results over the tested 1-hour period. The ratio temporarily drops for several large file transfers, but Lsync makes 90% of nodes remain fully synchronized for 72% of the tested period while the other approaches do not reach the synchronization ratio 0.9. Figure 11 shows the distributions of the completion latency for 10 KB, 100 KB, and 1 MB files.

6.5 Lsync Contributing Factors

Lsync combines various techniques discussed in the paper, including node scheduling, node selection, workload division across server and overlay, and adaptive switching in remote nodes. We examine the contribution of each factor individually. We transfer CoBlitz web proxy executable file (600 KB) to all remote nodes as before. The results of these tests are shown in Figure 13. Variations of Lsync are shown with no scheduling or node selection (No Policy), with only node scheduling (Slow First Scheduling), with only node selection (Fast Node Selection), with only overlay mesh (Using Overlay), and with all factors enabled (Lsync).

At a high level, we see that the individual contributions are significant, reducing the synchronization latency by a factor of 4-5 versus having no intelligence in the system. We see that performing scheduling improves the completion time for every target ratio, but that intelligent node selection is more critical at lower ratios. This result makes sense, since finding the fast nodes is more important when only a small fraction of the nodes are needed. However, when the ratio is high and even slower nodes are being included in the synchronization process, scheduling is needed to mask the effects of the slow nodes on the latency. The CDN is slow at low target ratios, and becomes comparable to the best end-to-end synchronization latency at high ratios due to its scalable file transfers. However, it shows the worst latency with high variations at the target ratio of 0.98 because some nodes experience performance problems in the overlay.

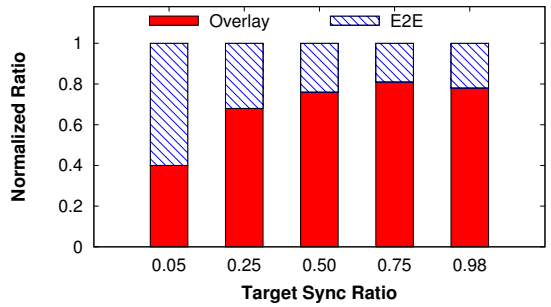


Figure 14: Division of Nodes in Lsync – We see that the fraction of nodes served by overlay mesh changes across target ratios, and that the fraction is not monotonically changing with target ratio.

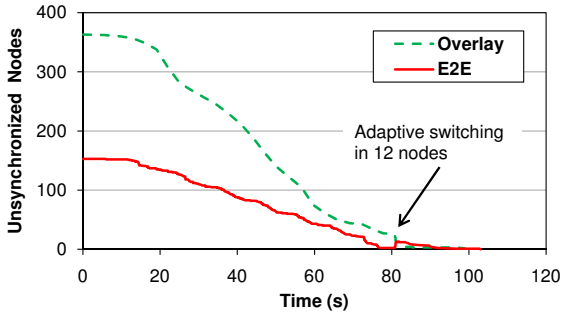


Figure 15: Adaptive Switching in Lsync – At 80 seconds, 12 nodes dynamically switch to end-to-end connections and finish downloading from the origin server.

Lsync combines all the factors in a manner to reduce the latency for all target ratios.

6.6 Nodes Division and Adaptive Switching

To see how Lsync adjusts workload across server and overlay mesh, we further analyze how nodes are divided into the overlay group and the end-to-end group. The nodes in the groups are selected so that both groups are expected to finish file transfer at the same time. In Figure 14, we plot the normalized sizes of the two groups during a large file (5MB) transfer. As the target synchronization ratio increases, a smaller fraction of nodes are served using end-to-end transfers. However, for the target ratio of 0.98, the overlay group's estimated completion time increases because of nodes with slow overlay connectivity. Therefore, the ratio for the origin server increases compared to the case of target ratio of 0.75.

Lsync makes adjustment at runtime, as can be seen in Figure 15, where we plot the number of pending nodes during file synchronization. The target synchronization ratio is 0.98, and r_{e2e}, the ratio of nodes for end-to-end connections, is 0.29. The two groups start downloading a 5 MB file through the overlay mesh and end-to-end connections respectively. At 80 seconds, however, 12 nodes in the overlay group detect that they are having unexpected overlay performance problems, and could negatively affect the completion time. The nodes dynamically switch to the end-to-end connections, and directly download the remaining content from the origin server. This behavior explains the small bump near 80 seconds – when these nodes switch from the overlay group to the end-to-end group, the number of unsynchronized end-to-end nodes increases.

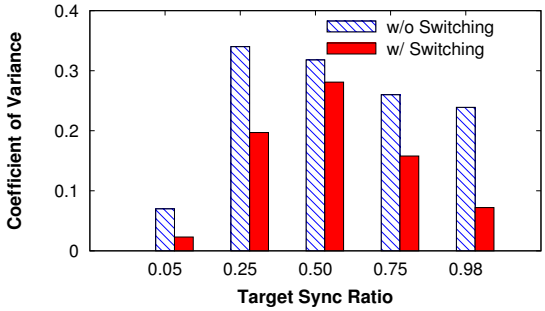


Figure 16: Stable File Transfers in Lsync – Adaptive switching in Lsync lowers variance of the latency.

During 10 repeated experiments, an average of 27 nodes dynamically switched to end-to-end connections. By assisting a few nodes in trouble, Lsync reduces the completion time by 16%. In addition to the reduced completion time, the adaptive switching in Lsync provides stable file transfers because it masks unpredictable variations in the overlay performance at runtime. Figure 16 plots the variations of the completion times with/without adaptive switching in Lsync. The vertical bars plot the coefficient of variance (normalized deviation) of completion times during repeated experiments. For every target ratio, Lsync shows smaller variations compared to the version with the switching disabled.

7 Related Work

CDNs and P2P systems [9, 16, 18, 19, 21] scalably distribute the content to a large number of clients. While these systems achieve bandwidth efficiency and load reduction at the origin server, they typically sacrifice startup time and total synchronization latency for smaller node groups. Likewise, peer-assisted swarm transfer systems [20] manage server's bandwidth using a global optimization, but they address bandwidth efficiency in multi-swarm environments, not latency.

Our work on managing latency in distributed systems is related in spirit to partial barrier [3] that is a relaxed synchronization barrier for loosely coupled networked systems. The proposed primitive provides dynamic knee detection in the node arrival process, and allows applications to release the barrier early before slow nodes arrive. Mantri [5] uses similar techniques to improve job completion time in Map-Reduce clusters.

Lsync aggressively uses available resources because it would otherwise remain unused. Another example is *dsync* [22] that aggressively draws data from multiple sources with varying performance. dsync schedules the resources based on the estimated cost and benefit of operations on each resource, and makes locally-optimal decisions, whereas Lsync tries to perform globally-optimal scheduling to reduce overall latency.

Gossip-based broadcast [8, 15] provides scalable and robust event dissemination to a large number of nodes. Our measurements demonstrate that the random peering strategy in the protocol helps reduce latency because slow nodes are likely to find nodes with the disseminated data after most fast nodes are synchronized. Lsync shows better latency than the gossip protocol because it targets the slow nodes from the beginning of file transfers, preventing the slow nodes from being the bottleneck.

8 Conclusion

Low-latency data dissemination is important for coordinating remote nodes in large-scale wide-area distributed systems, but the latency has been largely ignored in designing one-to-many file transfer systems. We found that the latency is highly variable in heterogeneous network environments, and many file transfer systems are suboptimal for the metric. To solve this problem, we have identified the sources of the latency and described techniques to reduce their impact on the latency (node scheduling, node selection, workload adjustment, and dynamic switching). We have presented Lsync that integrates all the techniques in a manner to minimize latency based on information available at runtime. In addition to standalone application, we expect that Lsync is useful to many wide-area distributed systems that need to coordinate the behavior of remote nodes with minimal latency.

9 Acknowledgments

We would like to thank our shepherd, Frank Dabek, as well as the anonymous reviewers. We also thank Michael Golightly, Sunghwan Ihm, and Anirudh Badam for useful discussion and their insightful comments on earlier drafts of this paper. This research was partially supported by the NSF Awards CNS-0615237, CNS-0916204, and KCA (Korea Communications Agency) in South Korea, KCA-2012-11913-05004.

References

[1] PlanetLab. http://www.planet-lab.org/.

[2] Akamai Inc. www.akamai.com/.

[3] J. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat. Loose synchronization for large-scale networked systems. In *Proceedings of USENIX ATC*, 2006.

[4] Amazon CloudFront. http://aws.amazon.com/cloudfront/.

[5] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in Map-Reduce clusters using Mantri. In *Proceedings of USENIX OSDI*, 2010.

[6] N. Bansal and M. Harchol-Balter. Analysis of SRPT scheduling: investigating unfairness. In *Proceedings of ACM SIGMETRICS*, 2001.

[7] O. Beaumont, N. Bonichon, and L. Eyraud-Dubois. Scheduling divisible workloads on heterogeneous platforms under bounded multi-port model. In *IEEE IPDPS*, 2008.

[8] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, May 1999.

[9] BitTorrent. http://bittorrent.com/.

[10] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth multicast in cooperative environments. In *Proceedings of ACM SOSP*, 2003.

[11] CoBlitz Inc. http://www.verivue.com/.

[12] A. Davis, J. Parikh, and W. E. Weihl. EdgeComputing: Extending enterprise applications to the edge of the internet. In *Proceedings of ACM WWW*, 2004.

[13] M. Deshpande, B. Xing, I. Lazardis, B. Hore, N. Venkatasubramanian, and S. Mehrotra. CREW: A gossip-based flash-dissemination system. In *IEEE ICDCS*, 2006.

[14] M. A. Eriksen. Trickle: A userland bandwidth shaper for unix-like systems. In *Proceedings of USENIX ATC*, 2005.

[15] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems*, November 2003.

[16] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *Proceedings of USENIX NSDI*, 2004.

[17] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proceedings of ACM SOSP*, 2003.

[18] E. Nygren, R. K. Sitaraman, and J. Sun. The Akamai network: A platform for high-performance internet applications. *ACM Operating Systems Review*, 2010.

[19] K. Park and V. S. Pai. Scale and performance in the CoBlitz large-file distribution service. In *Proceedings of USENIX NSDI*, 2006.

[20] R. S. Peterson, B. Wong, and E. G. Sirer. A content propagation metric for efficient content distribution. In *Proceedings of ACM SIGCOMM*, 2011.

[21] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani. Do incentives build robustness in BitTorrent? In *Proceedings of USENIX NSDI*, 2007.

[22] H. Pucha, M. Kaminsky, D. G. Andersen, and M. A. Kozuch. Adaptive file transfers for diverse environments. In *Proceedings of USENIX ATC*, 2008.

[23] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Harvard University, 1981.

[24] Riverbed Technology Inc. http://www.riverbed.com/.

[25] SETI@home. http://setiathome.berkeley.edu/.

[26] A. Sherman, P. A. Lisiecki, A. Berkheimer, and J. Wein. ACMS: The Akamai configuration management system. In *Proceedings of USENIX NSDI*, 2005.

[27] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, The Australian National University, 1999.

Generating Realistic Datasets for Deduplication Analysis

Vasily Tarasov[1], Amar Mudrankit[1], Will Buik[2], Philip Shilane[3], Geoff Kuenning[2], Erez Zadok[1]
[1]*Stony Brook University,* [2]*Harvey Mudd College, and* [3]*EMC Corporation*

Abstract

Deduplication is a popular component of modern storage systems, with a wide variety of approaches. Unlike traditional storage systems, deduplication performance depends on data content as well as access patterns and meta-data characteristics. Most datasets that have been used to evaluate deduplication systems are either unrepresentative, or unavailable due to privacy issues, preventing easy comparison of competing algorithms. Understanding how both content and meta-data evolve is critical to the realistic evaluation of deduplication systems.

We developed a generic model of file system changes based on properties measured on terabytes of real, diverse storage systems. Our model plugs into a generic framework for emulating file system changes. Building on observations from specific environments, the model can generate an initial file system followed by ongoing modifications that emulate the distribution of duplicates and file sizes, realistic changes to existing files, and file system growth. In our experiments we were able to generate a 4TB dataset within 13 hours on a machine with a single disk drive. The relative error of emulated parameters depends on the model size but remains within 15% of real-world observations.

1 Introduction

The amount of data that enterprises need to store increases faster than prices drop, causing businesses to spend ever more on storage. One way to reduce costs is deduplication, in which repeated data is replaced by references to a unique copy; this approach is effective in cases where data is highly redundant [11,15,17]. For example, typical backups contain copies of the same files captured at different times, resulting in deduplication ratios as high as 95% [9]. Likewise, virtualized environments often store similar virtual machines [11]. Deduplication can be useful even in primary storage [15], because users often share similar data such as common project files or recordings of popular songs.

The significant space savings offered by deduplication have made it an almost mandatory part of the modern enterprise storage stack [5,16]. But there are many variations in how deduplication is implemented and which optimizations are applied. Because of this variety and the large number of recently published papers in the area, it is important to be able to accurately compare the performance of deduplication systems.

The standard approach to deduplication is to divide the data into *chunks*, hash them, and look up the result in an index. Hashing is straightforward; chunking is well understood but sensitive to parameter settings. The indexing step is the most challenging because of the immense number of chunks found in real systems.

The chunking parameters and indexing method lead to three primary evaluation criteria for deduplication systems: (1) space savings, (2) performance (throughput and latency), and (3) resource usage (disk, CPU, and memory). All three metrics are affected by the data used for the evaluation and the specific hardware configuration. Although previous storage systems could be evaluated based only on the I/O operations issued, deduplication systems need the actual content (or a realistic re-creation) to exercise caching and index structures.

Datasets used in deduplication research can be roughly classified into two categories. (1) Real data from customers or users, which has the advantage of representing actual workloads [6, 15]. However, most such data is restricted and has not been released for comparative studies. (2) Data derived from publicly available releases of software sources or binaries [10, 24]. But such data cannot be considered as representative of the general user population. As a result, neither academia nor industry have wide access to representative datasets for unbiased comparison of deduplication systems.

We created a framework for *controllable data generation*, suitable for evaluating deduplication systems. Our dataset generator operates at the file-system level, a common denominator in most deduplication systems: even block- and network-level deduplicators often process file-system data. Our generator produces an initial file system image or uses an existing file system as a starting point. It then *mutates* the file system according to a *mutation profile*. To create profiles, we analyzed data and meta-data changes in several public and private datasets: home directories, system logs, email and Web servers, and a version control repository. The total size of our datasets approaches 10TB; the sum of observation periods exceeds one year, with the longest single dataset exceeding 6 months' worth of recordings.

Our framework is versatile, modular, and efficient. We use an in-memory file system tree that can be populated and mutated using a series of composable modules. Researchers can easily customize modules to emulate file system changes they observe. After all appropriate mutations are done, the in-memory tree can be quickly written to disk. For example, we generated a 4TB file system on a machine with a single drive in only 13 hours, 12 of which were spent writing data to the drive.

2 Previous Datasets

To quantify the lack of readily available and representative datasets, we surveyed 33 deduplication papers published in major conferences in 2000–2011: ten papers were Usenix ATC, ten in Usenix FAST, four in SYSTOR, two in IEEE MSST, and the remaining seven elsewhere. We classified 120 datasets used in these papers as: (1) Private datasets accessible only to particular authors; (2) Public datasets which are hard or impossible to reproduce (e.g., CNN web-site snapshots on certain dates); (3) Artificially synthesized datasets; and (4) Public datasets that are easily reproducible by anyone.

We found that 29 papers (89%) used *at least one* private dataset for evaluation. The remaining four papers (11%) used artificially synthesized datasets, but details of the synthesis were omitted. This makes it nearly impossible to fairly compare many papers among the 33 surveyed. Across all datasets, 64 (53%) were private, 17 (14%) were public but hard to reproduce, and 11 (9%) were synthetic datasets without generation details. In total, 76% of the datasets were unusable for cross-system evaluation. Of the 28 datasets (24%) we characterized as public, twenty were smaller than 1GB in logical size, much too small to evaluate any real deduplication system. The remaining eight datasets contained various operating system distributions in different formats: installed, ISO, or VM images.

Clearly, the few publicly available datasets do not adequately represent the entirety of real-world information. But releasing large real datasets is challenging for privacy reasons, and the sheer size of such datasets makes them unwieldy to distribute. Some researchers have suggested releasing hashes of files or file data rather than the data itself, to reduce the overall size of the released information and to avoid leaking private information. Unfortunately, hashes alone are insufficient: much effort goes into chunking algorithms, and there is no clear winning deduplication strategy because it often depends on the input data and workload being deduplicated.

3 Emulation Framework

In this section we first explain the generic approach we took for dataset generation and justify why it reflects many real-world situations. We then present the main components of our framework and their interactions. For the rest of the paper, we use the term *meta-data* to refer to the file system name-space (file names, types, sizes, directory depths, etc.), while *content* refers to the actual data within the files.

3.1 Generation Methods

Real-life file systems evolve over time as users and applications create, delete, copy, modify, and back up files. This activity produces several kinds of correlated information. Examples include 1) Identical downloaded files; 2) Users making copies by hand; 3) Source-control systems making copies; 4) Copies edited and modified by users and applications; 5) Full and partial backups repeatedly preserving the same files; and 6) Applications creating standard headers, footers, and templates.

To emulate real-world activity, one must account for all these sources of duplication. One option would be to carefully construct a statistical model that generates duplicate content. But it is difficult to build a statistics-driven system that can produce correlated output of the type needed for this project. We considered directly generating a file system containing duplicate content, but rejected the approach as impractical and non-scalable.

Instead, we emulate the evolution of real file systems. We begin with a simulated *initial snapshot* of the file system at a given time. (We use the term "snapshot" to refer to the complete state of a file system; our usage is distinct from the copy-on-write snapshotting technology available in some systems.) The initial snapshot can be based on a live file system or can be artificially generated by a system such as Impressions [1]. In either case, we *evolve* the snapshot over time by applying *mutations* that simulate the activities that generate both unique and duplicate content. Because our evolution is based on the way real users and applications change file systems, our approach is able to generate file systems and backup streams that accurately simulate real-world conditions, while offering the researcher the flexibility to tune various parameters to match a given situation.

Our mutation process can operate on file systems in two dimensions: space and time. The "space" dimension is equivalent to a single snapshot, and is useful to emulate deduplication in primary storage (e.g., if two users each have an identical copy of the same file). "Time" is equivalent to backup workloads, which are very common in deduplication systems, because snapshots are taken within some pre-defined interval (e.g., one day). Virtualized environments exhibit both dimensions, since users often create multiple virtual machines (VMs) with identical file systems that diverge over time because they are used for different purposes. Our system lets researchers create mutators for representative VM user classes and generate appropriately evolved file systems. Our system's ability to support logical changes in both space and time lets it evaluate deduplication for all major use cases: primary storage, backup, and virtualized environments.

3.2 Fstree Objects

Deduplication is usually applied to large datasets with hundreds of GB per snapshot and dozens of snapshots. Generating and repeatedly mutating a large file system would be unacceptably slow, so our framework performs

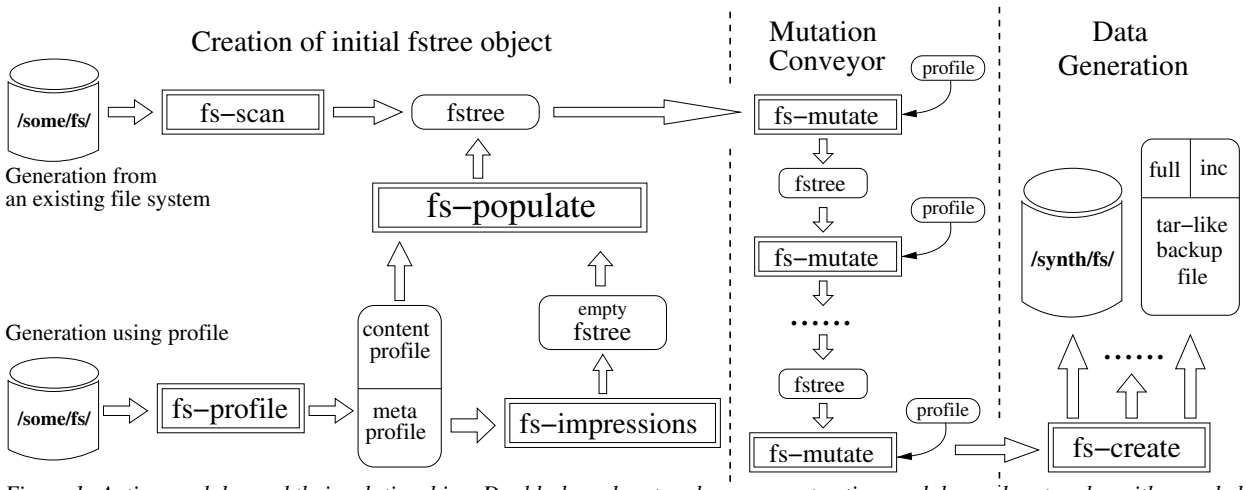


Figure 1: Action modules and their relationships. Double-boxed rectangles represent action modules and rectangles with rounded corners designate fstrees and other inputs and outputs.

most of its work without I/O. Output happens only at the end of the cycle when the actual file system is created.

To avoid excess I/O, we use a small in-memory representation—an *fstree*—that stores only the information needed for file system generation. This idea is borrowed from the design of Filebench [8]. The fstree contains pointers to *directory* and *file* objects. Each directory tracks its parent and a list of its files and subdirectories. The file object does not store the file's complete content; instead, we keep a list of its logical *chunks*, each of which has an identifier that corresponds to (but is not identical to) its deduplication hash. We later use the identifier to generate unique content for the chunk. We use only 4 bytes for a chunk identifier, allowing up to 2^{32} unique chunks. Assuming a 50% deduplication ratio and a 4KB average chunk size, this can represent 32TB of storage. Note that a single fstree normally represents a single snapshot, so 32TB is enough for most modern datasets. For larger datasets, the identifier field can easily be expanded.

To save memory, we do not track per-object user or group IDs, permissions, or other properties. If this information is needed in a certain model (e.g., if some users modify their files more often than others), all objects have a variable-sized private section that can store any information required by a particular emulation model.

The total size of the fstree depends on the number of files, directories, and logical chunks. File, directory, and chunk objects are 29, 36, and 20 bytes, respectively. Representing a 2TB file system in which the average file was 16KB and the average directory held ten files would require 9GB of RAM. A server with 64GB could thus generate realistic 14TB file systems. Note that this is the size of a *single* snapshot, and in many deduplication studies one wants to look at 2–3 months worth of daily backups. In this case, one would write a snapshot after each fstree mutation and then continue with the same in-memory fstree. In such a scenario, our system is capable of producing datasets of much larger sizes; e.g., for 90 full backups we could generate 1.2PB of test data.

Our experience has shown that it is often useful to save fstree objects (the object, not the full file system) to persistent storage. This allows us to reuse an fstree in different ways, e.g., representing the behavior of different users in a multi-tenant cloud environment. We designed the fstree so that it can be efficiently serialized to or from disk using only a single sequential I/O. Thus it takes less than two minutes to save or load a 9GB fstree on a modern 100MB/sec disk drive. Using a disk array can make this even faster.

3.3 Fstree Action Modules

An fstree represents a static image of a file system tree—a snapshot. Our framework defines several operations on fstrees, which are implemented by pluggable *action modules*; Figure 1 demonstrates their relationships. Double-boxed rectangles represent action modules; rounded ones designate inputs and outputs.

FS-SCAN. One way to obtain an initial fstree object (to be synthetically modified later) is to scan an existing file system. The FS-SCAN module does this: it scans content and meta-data, creates file, directory, and chunk objects, and populates per-file chunk lists. Different implementations of this module can collect different levels of detail about a file system, such as recognizing or ignoring symlinks, hardlinks, or sparse files, storing or skipping file permissions, using different chunking algorithms, etc.

FS-PROFILE, FS-IMPRESSIONS, and FS-POPULATE. Often, an initial file system is not available, or cannot be released even in the form of an fstree due to sensitive data. FS-PROFILE, FS-IMPRESSIONS, and FS-POPULATE address this problem. FS-PROFILE is similar to FS-SCAN, but does not collect such detailed information, instead gathering only a statistical profile. The spe-

Name	Total size (GB)	Total files (thousands)	Snapshots & period	Avg. snapshot size (GB)	Avg. number of files in a snapshot (thousands)
Kernels (Linux 2.6.0–2.6.39)	13	903	40	0.3	23
CentOS (5.0–5.7)	36	1,559	8	4.5	195
Home	3,482	15,352	15 weekly	227	1,023
MacOS	4,080	83,220	71 daily	59	1,173
System Logs	626	2,672	8 weekly	78	334
Sources	1,331	1,112	8 weekly	162	139

Table 1: Summary of analyzed datasets.

cific information collected depends on the implementation, but we assume it does not reveal sensitive data. We distinguish sub-parts: the *meta profile*, which contains statistics about the meta-data, and the *content profile*.

Several existing tools can generate a static file system image based on a meta-data profile [1, 8], and any of these can be reused by our system. A popular option is Impressions [1], which we modified to produce an fstree object instead of a file system image (FS-IMPRESSIONS). This fstree object is *empty*, meaning it contains no information about file contents. FS-POPULATE fills an empty fstree by creating chunks based on the content profile. Our current implementation takes the distribution of duplicates as a parameter; more sophisticated versions are future work.

The left part of Figure 1 depicts the two current options for creating initial fstrees. This paper focuses on the mutation module (below).

FS-MUTATE. FS-MUTATE is a key component of our approach. It mutates the fstree according to the changes observed in a real environment. Usually it iterates over all files and directories in the fstree and deletes, creates, or modifies them. A single mutation can represent weekly, daily, or hourly changes; updates produced by one or more users; etc. FS-MUTATE modules can be chained as shown in Figure 1 to represent multiple changes corresponding to different users, different times, etc. Usually, a mutation module is controlled by a parameterized profile based on real-world observations. The profile can also be chosen to allow micro-benchmarking, such as varying the percentage of unique chunks to observe changes in deduplication behavior. In addition, if a profile characterizes the changes between an empty file system and a populated one, FS-MUTATE can be used to generate an initial file system snapshot.

FS-CREATE. After all mutations are performed, FS-CREATE generates a final dataset in the form needed by a particular deduplication system. In the most common case, FS-CREATE produces a file system by walking through all objects, creating the corresponding directories and files, and generating file contents based on the chunk identifiers. Content generation is implementation-specific; for example, contents might depend on the file type or on an entropy level. The important property to preserve is that the same chunk identifiers result in the same content, and different chunk identifiers produce different content. FS-CREATE could also generate tar-like files for input to a backup system, which can be significantly faster than creating a complete file system because it can use sequential writes. FS-CREATE could also generate only the files that have changed since the previous snapshot, emulating data coming from an incremental backup.

4 Datasets Analyzed

To create a specific implementation of the framework modules, we analyzed file system changes in six different datasets; in each case, we used FS-SCAN to collect hashes and file system tree characteristics. We chose two commonly used public datasets, two collected locally, and two originally presented by Dong et al. [6].

Table 1 describes important characteristics of our six datasets: total size, number of files, and per-snapshot averages. Our largest dataset, MacOS, is 4TB in size and has 83 million files spanning 71 days of snapshots.

Kernels: Unpacked Linux kernel sources from version 2.6.0 to version 2.6.39.

CentOS: Complete installations of eight different releases of the CentOS Linux distribution from version 5.0 to 5.7.

Home: Weekly snapshots of students' home directories from a shared file system. The files consisted of source code, binaries, office documents, virtual machine images, and miscellaneous files.

MacOS: A Mac OS X Enterprise Server that hosts various services for our research group: email, mailing lists, Web-servers, wiki, Bugzilla, CUPS server, and an RT trouble-ticketing server.

System Logs: Weekly unpacked backups of a server's `/var` directory, mostly consisting of emails stored by a list server.

Sources: Weekly unpacked backups of source code and change logs from a Perforce version control repository.

Of course, different environments can produce significantly different datasets. For that reason, our design is flexible, and our prototype modules are parameterized by profiles that describe the characteristics of a particular dataset's changes. If necessary, other researchers can use our profile collector to gather appropriate distri-

butions, or implement a different FS-MUTATE model to express the changes observed in a specific environment.

For the datasets that we analyzed, we will release all profiles and module implementations publicly. We expect that future papers following this project will also publish their profiles and mutation module implementations, especially when privacy concerns prevent the release of the whole dataset. This will allow the community to reproduce results and better compare one deduplication system to another.

5 Module Implementations

There are many ways to implement our framework's modules. Each corresponds to a model that describes a dataset's behavior in a certain environment. An ideal model should capture the characteristics that most affect the behavior of a deduplication system. In this section we first explore the space of parameters that can affect the performance of a deduplication system, and then present a model for emulating our datasets' behavior. Our implementation can be downloaded from *https://avatar.fsl.cs.sunysb.edu/groups/deduplicationpublic/*.

5.1 Space Characteristics

Both content and meta-data characteristics are important for accurate evaluation of deduplication systems. Figure 2 shows a rough classification of relevant dataset characteristics. The list of properties in this section is not intended to be complete, but rather to demonstrate a variety of parameters that it might make sense to model.

Previous research has primarily focused on characterizing *static* file system snapshots [1]. Instead, we are interested in characterizing the file system's *dynamic* properties (both content and meta-data). Extending the analysis to multiple snapshots can give us information about file deletions, creations, and modifications. This in turn will reflect on the properties of static snapshots.

Any deduplication solution divides a dataset into chunks of fixed or variable size, indexes their hashes, and compares new incoming chunks against the index. If a new hash is already present, the duplicate chunk is discarded and a mapping that allows the required data to be located later is updated.

Therefore, the total number of chunks and the number of unique chunks in a dataset affects the system's performance. The performance of some data structures used in deduplication systems also depends on the distribution of duplicates, including the percentage of chunks with a certain number of duplicates and even the ordering of duplicates. E.g., it is faster to keep the index of hashes in RAM, but for large datasets a RAM index may be economically infeasible. Thus, many deduplication systems use sophisticated index caches and Bloom filters [25] to reduce RAM costs, complicating performance analysis.

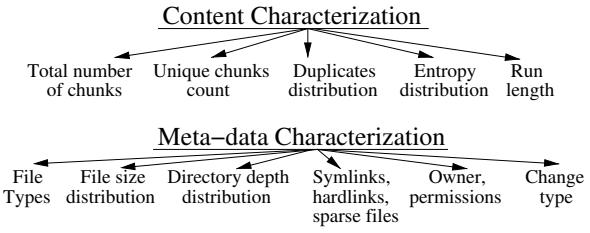


Figure 2: Content and meta-data characteristics of file systems that are relevant to deduplication system performance.

For many systems, it is also important to capture the entropy distribution inside the chunks, because most deduplication systems support local chunk compression to further reduce space. Compression can be enabled or disabled intelligently depending on the data type [12].

A deduplication system's performance depends not only on content, but also on the file system's *meta-data*. When one measures the performance of a conventional file system (without deduplication), the file size distribution and directory depth strongly impact the results [2]. Deduplication is sometimes an addition to existing conventional storage, in which case file sizes and directory depth will also affect the overall system performance.

The run lengths of unique or duplicated chunks can also be relevant. If unique chunks follow each other closely (in space and time), the storage system's I/O queues can fill up and throughput can drop. Run lengths depend on the ways files are modified: pure extension, as in log files; simple insertion, as for some text files; or complete rewrites, as in many binary files. Run lengths can also be indirectly affected by file size distributions, because it often happens that only a few files in the dataset change from one backup to another, and the distance between changed chunks within a backup stream depends on the sizes of the unchanged files.

Content-aware deduplication systems sometimes use the file header to detect file types and improve chunking; others use file owners or permissions to adjust their deduplication algorithms. Finally, symlinks, hardlinks, and sparse files are a rudimentary form of deduplication, and their presence in a dataset can affect deduplication ratios.

Dependencies. An additional issue is that many of the parameters mentioned above depend on each other, so considering their statistical distributions independently is not possible. For example, imagine that emulating the changes to a specific snapshot requires removing N files. We also want the total number of chunks to be realistic, so we need to remove files of an appropriate size. Moreover, the distribution of duplicates needs to be preserved, so the files that are removed should contain the appropriate number of unique and duplicated chunks. Preserving such dependencies is important, and our FS-MUTATE implementation (presented next) allows that.

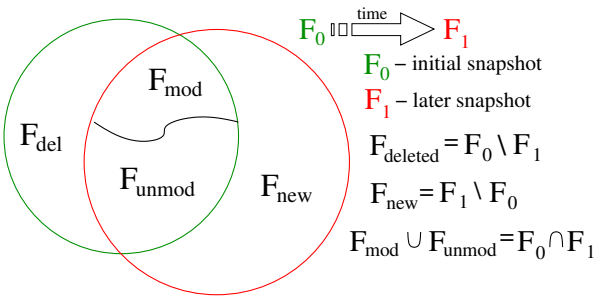


Figure 3: Classification of files. F_0 and F_1 are files from two subsequent snapshots.

5.2 Markov & Distribution (M&D) Model

We call our model *M&D* because it is based on two abstractions: a Markov model for classifying file changes, and a multi-dimensional distribution for representing statistical dependencies between file characteristics.

Markov model. Suppose we have two snapshots of a file system taken at two points in time: F_0 and F_1. We classify files in F_0 and F_1 into four sets: 1) F_{del}: files that exist in F_0, but are missing in F_1. 2) F_{new}: files that exist in F_1, but are missing in F_0. 3) F_{mod}: files that exist in both F_0 and F_1, but were modified. 4) F_{unmod}: files in F_0 and F_1 that were not modified. The relationship between these sets is depicted in Figure 3. In our study, we identify files by their full pathname, i.e., a file in the second snapshot with the same pathname as one in the first is assumed to be a later version of the same file.

Analysis of our datasets showed that the file sets defined above remain relatively stable. Files that were unmodified between snapshots $F_0 \rightarrow F_1$ tended to remain unmodified between snapshots $F_1 \rightarrow F_2$. However, files still migrate between sets, with different rates for different datasets. To capture such behavior we use the Markov model depicted in Figure 4. Each file in the fstree has a state assigned to it in accordance with the classification defined earlier. In the fstree representing the first snapshot, all files have the New state. Then, during mutation, the file states change with precalculated probabilities that have been extracted by looking at a window of three real snapshots, covering two file transitions: between the first and second snapshots and between the second and third ones. This is the minimum required to allow us to calculate conditional probabilities for the Markov model. For example, if some file is modified between snapshots $F_0 \rightarrow F_1$ and is also modified in $F_1 \rightarrow F_2$, then this is a Modified→Modified (MM) transition. Counting the number of MM transitions among the total number of state transitions allows us to compute the corresponding probability; we did this for each possible transition.

Some transitions, such as Deleted→New (DN), may seem counterintuitive. However, some files are recreated after being deleted, producing nonzero probabilities for this transition. Similarly, if a file is renamed or moved, it will be counted as two transitions: a removal and a creation. In this case, we allocate duplicated chunks to the new file in a later stage.

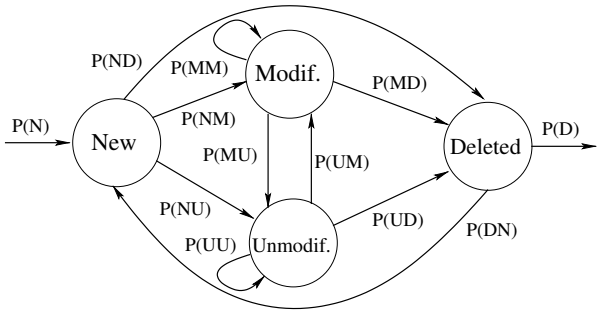


Figure 4: Markov model for handling file states. State transitions are denoted by the first letters of the source and destination states. For example, NM denotes a New→Modified transition and P(NM) is the transition's probability.

The Markov model allows us to accurately capture the rates of file appearance, deletion, and modification in the trace. Table 2 presents the average transition probabilities observed for our datasets. As mentioned earlier, in all datasets files often remain Unchanged, and thus the probabilities of UU transitions are high. The chances for a changed file to be re-modified are around 50% for many of our datasets. The probabilities for many other transitions vary significantly across different datasets.

Multi-dimensional distribution. When we analyzed real snapshots, we collected three multi-dimensional file distributions: $M_{del}(p_1, ..., p_{n_{del}})$, $M_{new}(p_1, ..., p_{n_{new}})$, and $M_{mod}(p_1, ..., p_{n_{mod}})$ for deleted, new, and modified files, respectively. The parameters of these distributions (p_1, ..., p_n) represent the characteristics of the files that were deleted, created, or modified. As described in Section 5.1, many factors affect deduplication. In this work, we selected several that we deemed most relevant for a generic deduplication system. However, the organization of our FS-MUTATE module allows the list of emulated characteristics to be easily extended.

All three distributions include these parameters:

depth: directory depth of a file;
ext: file extension;
size: file size (in chunks):
uniq: the number of chunks in a file that are not present in the previous snapshot (i.e., unique chunks);
dup1: the number of chunks in a file that have only one duplicate in the entire previous snapshot; and
dup2: the number of chunks in a file that occur exactly twice in the entire previous snapshot.

We consider only the chunks that occur up to 3 times in a snapshot because in all our snapshots these chunks constituted more than 96% of all chunks.

During mutation, we use the distribution of new files:

$$M_{new}(depth, ext, size, uniq, dup1, dup2)$$

Dataset	N	NM	NU	ND	MU	MD	MM	UM	UD	UU	DN	D
Kernels	5	32	65	3	49	3	48	17	3	80	1	3
CentOS	13	4	22	74	43	2	55	4	1	95	1	10
Home	4	2	78	20	54	10	36	0.14	0.35	99.51	6	0.50
MacOS	0.1	11	78	11	37.46	0.03	62.51	0.05	0.03	99.92	1	0.03
System Logs	2	9	90	1	44.40	0.18	55.42	0.03	0.01	99.06	4	0.02
Sources	0.2	7	88	5	58.76	0.04	41.20	0.07	0	99.93	0	0.01

Table 2: Probabilities (in percents) of file state transitions for different datasets. N: new file appearance. D: file deletion. NM: New→Modified transition. NU: New→Unmodified transition. ND: New→Deleted transition, etc.

to create the required number of files with the appropriate properties. E.g., if $M_{new}(2, \text{“.c”}, 7, 3, 1, 1)$ equals four, then FS-MUTATE creates four files with a “.c” extension at directory depth two. The size of the created files is seven chunks, of which three are unique, one has a single duplicate, and one has two duplicates across the entire snapshot. The hashes for the remaining two chunks are selected using a per-snapshot (not per-file) distribution of duplicates, which is collected during analysis along with M_{new}. Recall that FS-MUTATE does not generate the content of the chunks, but only their hashes. Later, during on-disk snapshot creation, FS-CREATE will generate the content based on the hashes.

When selecting files for deletion, FS-MUTATE uses the deleted-files distribution:

$$M_{del}(depth, ext, size, uniq, dup1, dup2, state)$$

This contains an additional parameter—*state*—that allows us to elegantly incorporate a Markov model in the distribution. The value of this parameter can be one of the Markov states New, Modified, Unmodified, or Deleted; we maintain the state of each file within the fstree. A file is created in the New state; later, if FS-MUTATE modifies it, its state is changed to Modified; otherwise it becomes Unmodified. When FS-MUTATE selects files for deletion, it limits its search to files in the state given by the corresponding M_{del} entry. For example, if $M_{del}(2, \text{“.c”}, 7, 3, 1, 1, \textit{“Modified”})$ equals one, then FS-MUTATE tries to delete a single file in the Modified state (all other parameters should match as well).

To select files for modification, FS-MUTATE uses the M_{mod} distribution, which has the same parameters as M_{del}. But unlike deleted files, FS-MUTATE needs to decide *how* to change the files. For every entry in M_{mod}, we keep a list of *change descriptors*, each of which contains the file's characteristics *after* modification:

1. File size (in chunks);
2. The number of unique chunks (here and in the two items below, duplicates are counted against the entire snapshot);
3. The number of chunks with one duplicate;
4. The number of chunks with two duplicates; and
5. Change pattern.

All parameters except the last are self-explanatory. The change pattern encodes the way a file was modified. We currently support the following three options: B—the file was modified in the beginning (this usually corresponds to prepend); E—the file was modified at the end (corresponds to file extension or truncation); and M—the file was modified somewhere in the middle, which corresponds to the case when neither the first nor the last chunk were modified, but others have changed. We also support combinations of these patterns: *BE*, *BM*, *EM*, and *BEM*. To recognize the change pattern during analysis, we sample the corresponding chunks in the old and new files. Table 3 presents the average change patterns for different datasets. For all datasets the number of files modified in the beginning is high. This is a consequence of chunk-based analysis: files that are smaller than the chunk size contain a single chunk. Therefore, wherever small files are modified, the first (and only) chunk differs in two subsequent versions, which our analysis identifies as a change in the file's beginning. For the System Logs dataset, the number of files modified at the end is high because logs are usually appended. In the Sources dataset many files are modified in the middle, which corresponds to small patches in the code.

Dataset	B	E	M	BE	BM	ME	BEM
Kernels	52	8	7	14	5	3	11
CentOS	69	3	2	8	2	1	15
Home	38	3	8	10	11	1	29
MacOS	53	21	1	12	1	1	11
Sys. Logs	42	34	5	6	0	1	10
Sources	20	6	41	7	7	1	18

Table 3: Probabilities of the change patterns for different datasets (in percents).

We collect change descriptors and the M_{mod} distribution during the analysis phase. During mutation, when a file is selected for modification using M_{mod}, one of the aforementioned change descriptors is selected randomly and the appropriate changes are applied.

It is possible that the number of files that satisfy the distribution parameters is larger than the number that need to be deleted or modified. In this case, FS-MUTATE randomly selects files to operate on. If not enough files with the required properties are in the fstree, then FS-MUTATE tries to find the best match based on a simple

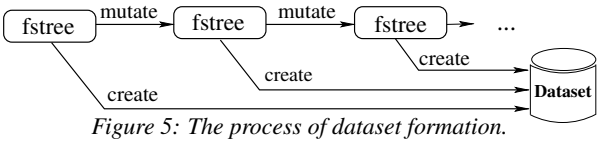


Figure 5: The process of dataset formation.

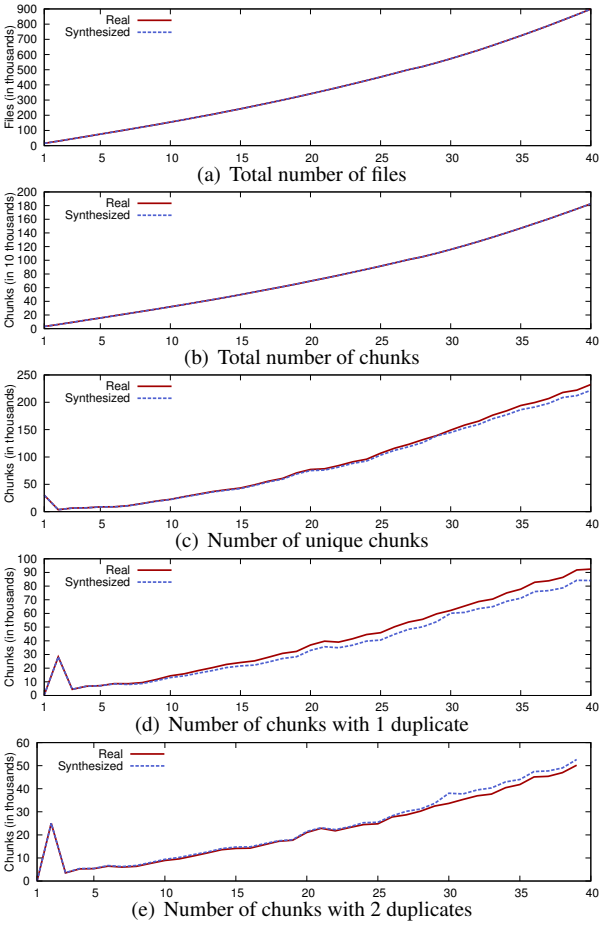


Figure 6: Emulated parameters for Kernels real and synthesized datasets as the number of snapshots in them increases.

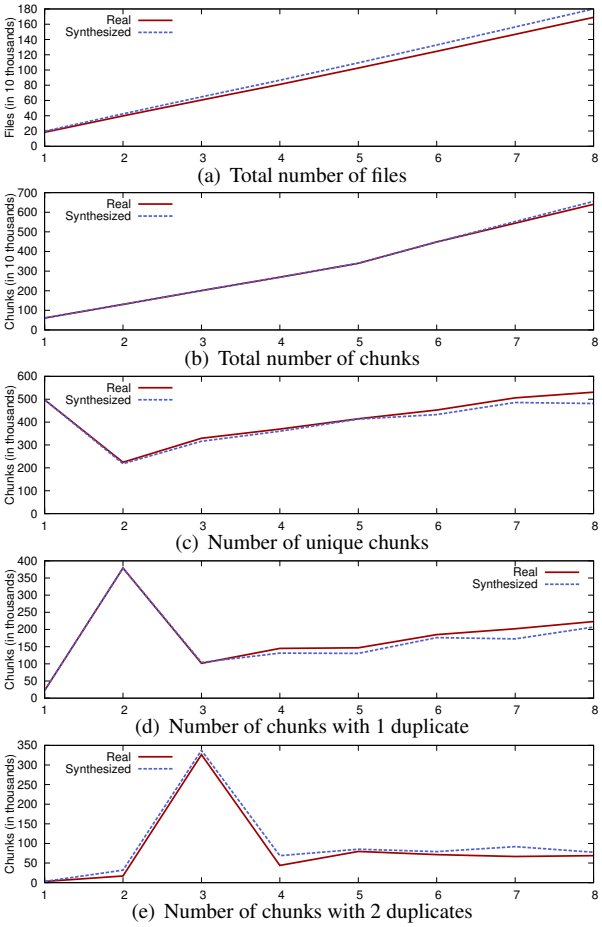


Figure 7: Emulated parameters for CentOS real and synthesized datasets as the number of snapshots in them increases.

heuristic: the file that matches most of the properties. Other definitions of best match are possible, and we plan to experiment with this parameter in the future.

Multi-dimensional distributions capture not only the statistical frequency of various parameters, but also their interdependencies. By adding more distribution dimensions, one can easily emulate other parameters.

Analysis. To create profiles for our datasets, we first scanned them using the FS-SCAN module mentioned previously. We use variable chunking with an 8KB average size; variable chunking is needed to properly detect the type of file change, since prepended data causes fixed-chunking systems to see a change in every chunk. We chose 8KB as a compromise between accuracy (smaller sizes are more accurate) and the speed of the analysis, mutation, and file system creation steps.

The information collected by FS-SCAN was loaded into a database; we then used SQL queries to extract distributions. The analysis of our smallest dataset (Kernels) took less than 2 hours, whereas the largest dataset (MacOS) took about 45 hours of wall-clock time on a single workstation. This analysis can be sped up by parallelizing it. However, since it needs to be done only once to extract a profile, a moderately lengthy computation is acceptable. Mutation and generation of a file system run much faster and are evaluated in Section 6. The size of the resulting profiles varied from 8KB to 300KB depending on the number of changes in the dataset.

Chunk generation. Our FS-CREATE implementation generates chunk content by maintaining a randomly generated buffer. Before writing a chunk to the disk, this buffer is XORed with the chunk ID to ensure that each ID produces a unique chunk and that duplicates have the same content. We currently do not preserve the chunk's entropy because our scan tool does not yet collect this data. FS-SCAN collects the size of every chunk, which is kept in the in-memory fstree object for use by FS-CREATE. New chunks in mutated snapshots have their size set by FS-MUTATE according to a per-snapshot chunk-size distribution. However, deduplication systems can use *any* chunk size that is larger than or equal to the one that FS-SCAN uses. In fact, sequences of identical chunks may appear in several subsequent snapshots. As these sequences of chunks are relatively long, any

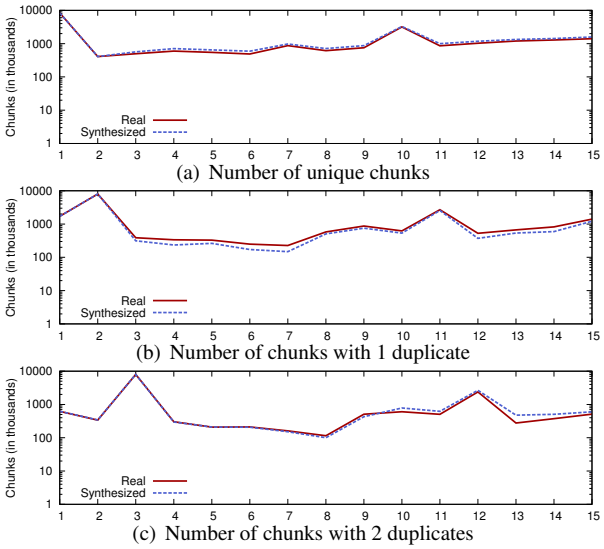


Figure 8: Emulated parameters for Homes real and synthesized datasets as the number of snapshots in them increases.

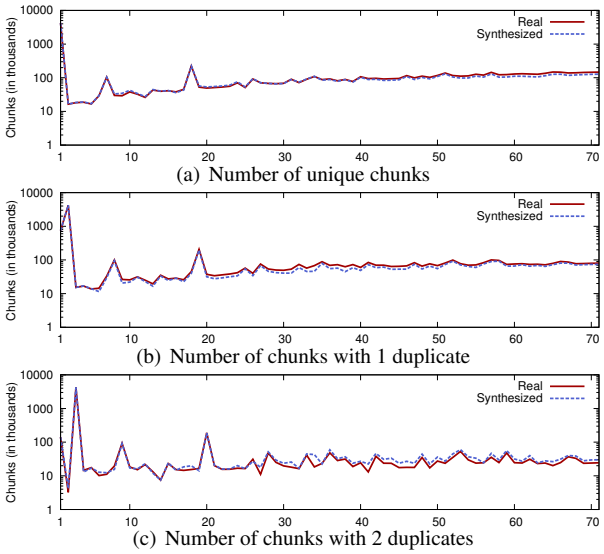


Figure 9: Emulated parameters for MacOS real and synthesized datasets as the number of snapshots in them increases.

chunking algorithm can detect an appropriate number of identical chunks across several snapshots.

Security guarantees. The FS-SCAN tool uses 48-bit fingerprints, which are prefixes of 16 byte MD5 hashes; this provides a good level of security, although we may be open to dictionary attacks. Stronger anonymization forms can be easily added in the future work.

6 Evaluation

We collected profiles for the datasets described in Section 4 and generated the same number of synthetic snapshots as the real datasets had, chaining different invocations of FS-MUTATE so that the output of one mutation served as input to the next. All synthesized snapshots together form a synthetic dataset that corresponds to the whole real dataset (Figure 5). We generated the initial fstree object by running FS-SCAN on the real file system. Each time a new snapshot was added, we measured the total files, total chunks, numbers of unique chunks and those that had one and two duplicates, directory depth, file size and file type distributions.

First, we evaluated the parameters that FS-MUTATE emulates. Figures 6–11 contain the graphs for the real and synthesized Kernels, CentOS, Homes, MacOS, System Logs, and Sources datasets, in order. The Y axis scale is linear for the Kernels and Sources datasets (Figures 6–7) and logarithmic for the others (Figures 8–11). We present file and chunk count graphs only for the Kernels and CentOS datasets. The relative error of these two parameters is less than 1% for all datasets, and the graphs look very similar: monotonic close-to-linear growth. The file count is insensitive to modification operations because files are not created or removed, which explains its high accuracy. The total chunk count

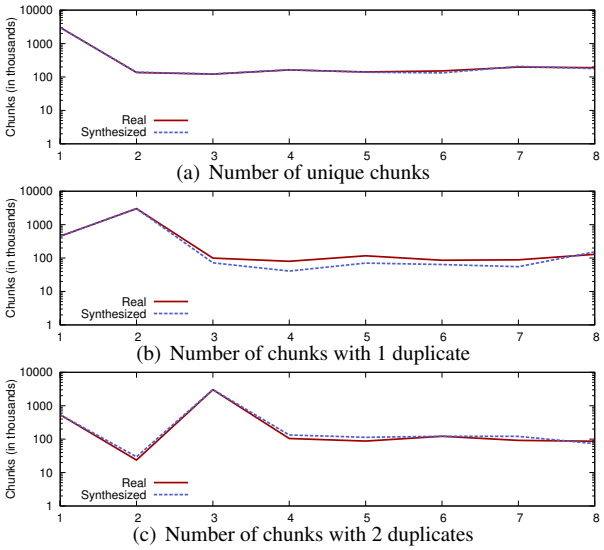
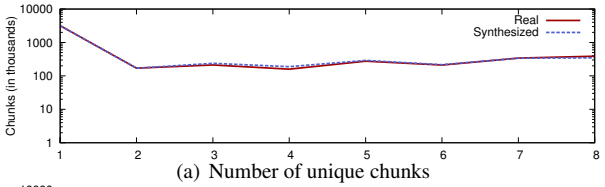


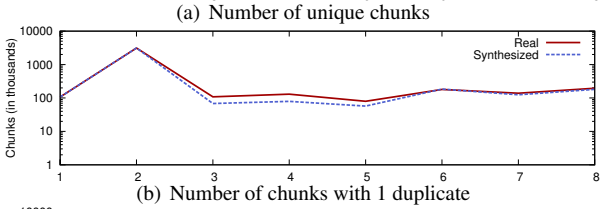
Figure 10: Emulated parameters for System Logs real and synthesized datasets as the number of snapshots in them increases.

is maintained because we carefully preserve file size during creation, modification, and deletion.

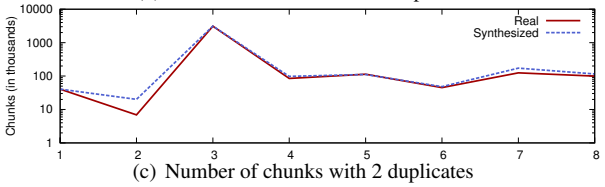
For all datasets the trends observed in the real data are closely followed by the synthesized data. However, certain discrepancies exist. Some of the steps in our FS-MUTATE module are random; e.g., the files deleted or modified are not precisely the same ones as in the real snapshot, but instead ones with similar properties. This means that our synthetic snapshots might not have the same files that would exist in the real snapshot. As a result, FS-MUTATE cannot find some files during the following mutations and so the best-match strategy is used, contributing to the *instantaneous* error of our method.



(a) Number of unique chunks

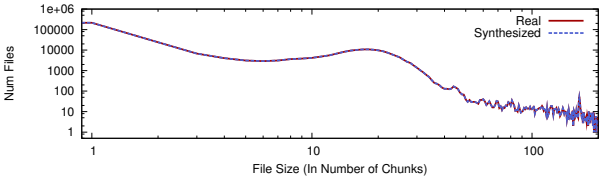


(b) Number of chunks with 1 duplicate

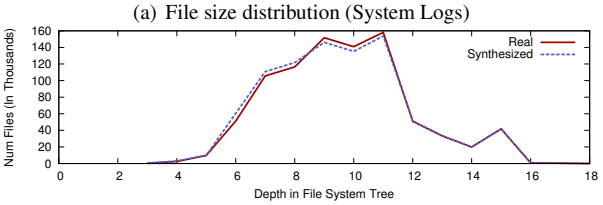


(c) Number of chunks with 2 duplicates

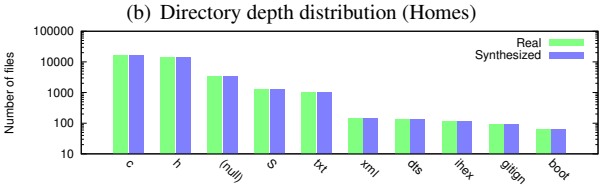
Figure 11: Emulated parameters for Sources real and synthesized datasets as the number of snapshots in them increases.



(a) File size distribution (System Logs)



(b) Directory depth distribution (Homes)



(c) File type distribution (Kernels)

Figure 12: File size, type, and directory depth distributions for different real and synthesized dataset.

However, because our random actions are controlled by the real statistics, the deviation is limited in the long run.

The graphs for unique chunks have an initial peak because there is only one snapshot at first, and there are not many duplicates in a single snapshot. As expected, this peak moves to the right in the graphs for chunks with one and two duplicates.

The Homes dataset has a second peak in all graphs around 10–12 snapshots (Figure 8). This point corresponds to two missing weekly snapshots. The first was missed due to a power outage; the second was missed because our scan did not recover properly from the power outage. As a result, the 10th snapshot contributes many more unique chunks in the dataset than the others.

The MacOS dataset contains daily, not weekly snapshots. Daily changes in the system are more sporadic than weekly ones: one day users and applications add a lot of new data, the next many files are copied, etc. Figure 9 therefore contains many small variations.

Table 4 shows the relative error for emulated parameters at the end of each run. Maximum deviation did not exceed 15% and averaged 6% for all parameters and datasets. We also analyzed the file size, type, and directory depth distributions in the final dataset. Figure 12 demonstrates these for several representative datasets. In all cases the accuracy was fairly high, within 2%.

The snapshots in our datasets change a lot. For example, the deduplication ratio is less than 5 in our Kernels dataset, even though the number of snapshots is 40. We expect the accuracy of our system to be higher for the datasets that change slower; for instance, datasets with identical snapshots are emulated without any error.

Performance. We measured the time of every mutation and creation operation in the experiments above.

Dataset	Files	Chunks	Unique chunks	1 Dup. chunks	2 Dup. chunks
Kernels	< 1	< 1	4	9	5
CentOS	6	2	9	7	11
Home	< 1	< 1	12	13	14
MacOS	< 1	< 1	4	9	4
Sys. Logs	< 1	< 1	6	15	15
Sources	< 1	< 1	10	8	13

Table 4: Relative error of emulated parameters after the final run for different datasets (in percents).

Dataset	Total size (GB)	Snapshots	Mutat. time	Creat. time	Total time
Kernels	13	40	30 sec	6 sec	5 min
CentOS	36	8	3 min	95 sec	13 min
Home	3,482	15	44 min	38 min	10 hr
MacOS	4,080	71	49 min	10 min	13 hr
Sys. Logs	626	8	14 min	4 hr	32 hr
Sources	1,331	8	21 min	4 hr	32 hr

Table 5: Times to mutate and generate data sets.

The Kernels, CentOS, Home, and MacOS experiments were conducted on a machine with an Intel Xeon X5680 3.3GHz CPU and 64GB of RAM. The snapshots were written to a single Seagate Savvio 15K RPM disk drive. For some datasets the disk drive could not hold all the snapshots, so we removed them after running FS-SCAN for accuracy analysis. Due to privacy constraints the System Logs and Sources experiments were run on a different machine with an AMD Opteron 2216 2.4GHz CPU, 32GB of RAM, and a Seagate Barracuda 7,200 RPM disk drive. Unfortunately, we had to share the second machine with a long-running job that periodically performed random disk reads.

Table 5 shows the *total* mutation time for all snap-

shots, the time to write a *single* snapshot to the disk, and the total time to perform all mutations plus write the whole dataset to the disk. The creation time includes the time to write to disk. For convenience the table also contains dataset sizes and snapshot counts.

Even for the largest dataset, we completed all mutations within one hour; dataset size is the major factor in mutation time. Creation time is mostly limited by the underlying system's performance: the creation throughput of the Home and MacOS datasets is almost twice that of Kernels and CentOS, because the average file size is 2–10× larger for the former datasets, exploiting the high sequential drive throughput. The creation time was significantly increased on the second system because of a slower disk drive (7,200RPM vs. 15KRPM) and the interfering job, contributing to the 32-hour run time.

For the datasets that can fit in RAM—CentOS and Kernels—we performed an additional FS-CREATE run so that it creates data on tmpfs. The throughput in both cases approached 1GB/sec, indicating that our chunk generation algorithm does not incur much overhead.

7 Related Work

A number of studies have characterized file system workloads using I/O traces [13, 19, for example] that contain information about all I/O requests observed during a certain period. The duration of a full trace is usually limited to several days, which makes it hard to analyze long-term file system changes. Trace-based studies typically focus on the dynamic properties of the workload, such as I/O size, read-to-write ratio, etc., rather than file content as is needed for deduplication studies.

Many papers have used snapshots to characterize various file system properties [2, 3, 20, 22]. With the exception of Agrawal et al.'s study [2], discussed below, the papers examine only a single snapshot, so only static properties can be extracted and analyzed. Because conventional file systems are sensitive to meta-data characteristics, snapshot-based studies focus on size distributions, directory depths or widths, and file types (derived from extensions). File and block lifetimes are analyzed based on timestamps [2, 3, 22]. Authors often discuss the correlation between file properties, e.g., size and type [3,20]. Several studies have proposed high-level explanations for file size distributions and designed models for synthesizing specific distributions [7, 20].

Less attention has been given to the analysis of long-term file system changes. Agrawal et al. examined the trends in file system characteristics from 2000–2004 [2]. The authors presented only meta-data evolution: file count, size, type, age, and directory width and depth.

Some researchers have worked on artificial file system aging [1, 21] to emulate the fragmentation encountered in real long-lived file systems. Our mutation module modifies the file system in RAM and thus does not emulate file system fragmentation. Modeling fragmentation can be added in the future if it proves to impact deduplication systems' performance significantly.

A number of newer studies characterized deduplication ratios for various datasets. Meyer and Bolosky studied content and meta-data in primary storage [15]. The authors collected file system content from over 800 computers and analyzed the deduplication ratios of different algorithms: whole-file, fixed chunking, and variable chunking. Several researchers characterized deduplication in backup storage [17, 23] and for virtual machine disk images [11, 14]. Chamness presented a model for storage-capacity planning that accounts for the number of duplicates in backups [4]. None of these projects attempted to synthesize datasets with realistic properties.

File system benchmarks usually create a test file system from scratch. For example, in Filebench [8] one can specify file size and directory depth distributions for the creation phase, but the data written is either all zeros or random. Agrawal et al. presented a more detailed attempt to approximate the distributions encountered in real-world file systems [1]. Again, no attention was given in their study to generating duplicated content.

8 Conclusions and Future Work

Researchers and companies evaluate deduplication with a variety of datasets that in most cases are private, unrepresentative, or small in size. As a result, the community lacks the resources needed for fair and versatile comparison. Our work has two key contributions.

First, we designed and implemented a generic framework that can emulate the formation of datasets in different scenarios. By implementing new mutation modules, organizations can expose the behavior of their internal datasets without releasing the actual data. Other groups can then regenerate comparable data and evaluate different deduplication solutions. Our framework is also suitable for controllable micro-benchmarking of deduplication solutions. It can generate arbitrarily large datasets while still preserving the original's relevant properties.

Second, we presented a specific implementation of the mutation module that emulates the behavior of several real-world datasets. To capture the meta-data and content characteristics of the datasets, we used a hybrid Markov and Distribution model that has a low error rate—less than 15% during 8 to 71 mutations for all datasets. We plan to release the tools and profiles described in this paper so that organizations can perform comparable studies of deduplication systems. These powerful tools will help both industry and research to make intelligent decisions when selecting the right deduplication solutions for their specific environments.

Future Work. Our specific implementation of the framework modules might not model all parameters that potentially impact the behavior of existing deduplication systems. We plan to conduct a study similar to Park et al. [18] to create a complete list of the dataset properties that impact deduplication systems. Although we can generate an initial file system snapshot using a specially collected profile for FS-MUTATE, such approach can be limiting. We plan to perform an extensive study on how to create initial fstree objects. Many deduplication systems perform local chunk compression to achieve even higher aggregate compression. We plan to incorporate into our framework a method for generating chunks with a realistic compression ratio distribution. Finally, we want to apply clustering techniques to detect trend lines so that more generic profiles can be designed.

Acknowledgments. We thank the anonymous Usenix reviewers and our shepherd for their help. This work was made possible in part thanks to NSF awards CCF-0937833 and CCF-0937854, a NetApp Faculty award, and an IBM Faculty award.

References

[1] N. Agrawal, A. C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau. Generating realistic impressions for file-system benchmarking. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST '09)*, 2009.

[2] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. In *Proceedings of the Fifth USENIX Conference on File and Storage Technologies (FAST '07)*, 2007.

[3] J. M. Bennett, M. A. Bauer, and D. Kinchlea. Characteristics of files in NFS environments. *ACM SIGSMALL/PC Notes*, 18(3-4):18–25, 1992.

[4] M. Chamness. Capacity forecasting in a backup storage environment. In *Proceedings of USENIX Large Installation System Administration Conference (LISA)*, 2011.

[5] EMC Corporation. EMC Centra: content addressed storage systems. Product description guide, 2004.

[6] W. Dong, F. Douglis, K. Li, H. Patterson, S. Reddy, and P. Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *Proceedings of the Nineth USENIX Conference on File and Storage Technologies (FAST '11)*, 2011.

[7] A. B. Downey. The structural cause of file size distributions. In *Proceedings of IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS)*, 2001.

[8] Filebench. *http://filebench.sourceforge.net*.

[9] Advanced Storage Products Group. Identifying the hidden risk of data deduplication: how the HYDRAstor solution proactively solves the problem. Technical Report WP103-3_0709, NEC Corporation of America, 2009.

[10] N. Jain, M. Dahlin, and R. Tewari. TAPER: tiered approach for eliminating redundancy in replica synchronization. In *Proceedings of the Fourth USENIX Conference on File and Storage Technologies (FAST '05)*, 2005.

[11] K. Jin and E. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of the Second ACM Israeli Experimental Systems Conference (SYSTOR '09)*, 2009.

[12] R. Kothiyal, V. Tarasov, P. Sehgal, and E. Zadok. Energy and performance evaluation of lossless file data compression on server systems. In *Proceedings of the Second ACM Israeli Experimental Systems Conference (SYSTOR '09)*, 2009.

[13] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the USENIX Annual Technical Conference (ATC '08)*, 2008.

[14] A. Liguori and E. Hensbergen. Experiences with content addressable storage and virtual disks. In *Proceedings of the Workshop on I/O Virtualization (WIOV)*, 2008.

[15] D. Meyer and W. Bolosky. A study of practical deduplication. In *Proceedings of the Nineth USENIX Conference on File and Storage Technologies (FAST '11)*, 2011.

[16] NetApp. NetApp deduplication for FAS. Deployment and implementation, 4th revision. Technical Report TR-3505, NetApp, 2008.

[17] N. Park and D. Lilja. Characterizing datasets for data deduplication in backup applications. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2010.

[18] N. Park, W. Xiao, K. Choi, and D. J. Lilja. A statistical evaluation of the impact of parameter selection on storage system benchmark. In *Proceedings of the 7th IEEE International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, 2011.

[19] D. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *Proceedings of the Annual USENIX Technical Conference*, 2000.

[20] M. Satyanarayanan. A study of file sizes and functional lifetimes. In *Proceedings of the 8th ACM Symposium on Operating System Principles (SOSP '81)*, 1981.

[21] K. A. Smith and M. I. Seltzer. File system aging—increasing the relevance of file system benchmarks. In *Proceedings of the 1997 International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1997)*, 1997.

[22] W. Vogels. File system usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, 1999.

[23] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of backup workloads in production systems. In *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST '12)*, 2012.

[24] W. Xia, H. Jiang, D. Feng, and Y. Hua. SiLo: A similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2011.

[25] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the Sixth USENIX Conference on File and Storage Technologies (FAST '08)*, 2008.

An Empirical Study of Memory Sharing in Virtual Machines

Sean Barker, Timothy Wood†, Prashant Shenoy, Ramesh Sitaraman
University of Massachusetts Amherst
† *The George Washington University*
[sbarker,shenoy,ramesh]@cs.umass.edu, timwood@gwu.edu

Abstract

Content-based page sharing is a technique often used in virtualized environments to reduce server memory requirements. Many systems have been proposed to capture the benefits of page sharing. However, there have been few analyses of page sharing in general, both considering its real-world utility and typical sources of sharing potential. We provide insight into this issue through an exploration and analysis of memory traces captured from real user machines and controlled virtual machines. First, we observe that absolute sharing levels (excluding zero pages) generally remain under 15%, contrasting with prior work that has often reported savings of 30% or more. Second, we find that sharing within individual machines often accounts for nearly all (>90%) of the sharing potential within a set of machines, with inter-machine sharing contributing only a small amount. Moreover, even small differences between machines significantly reduce what little inter-machine sharing might otherwise be possible. Third, we find that OS features like address space layout randomization can further diminish sharing potential. These findings both temper expectations of real-world sharing gains and suggest that sharing efforts may be equally effective if employed within the operating system of a single machine, rather than exclusively targeting groups of virtual machines.

1 Introduction

Many modern computing services are hosted in large-scale data centers, where many separate applications are served simultaneously on thousands of machines. In the Infrastructure-as-a-Service model (IaaS), exemplified by services such as Amazon EC2, data center operators rent machine resources to customers, who then operate their own applications on their rented machines.

To facilitate servicing multiple customers on single servers, providers have turned to *virtualization*, in which multiple virtual machines (VMs) operate independently but are collocated on a single physical server. In such a model, memory efficiency is a key consideration, as it is often the bottleneck resource that determines the number of customers that providers can service on individual physical servers. Most virtualized systems simply partition available physical memory between VMs. This results in a hard cap of the number of customer VMs possible per machine based on the amount of memory consumed by each VM. If each customer can be serviced using only a small amount of memory, more customers can potentially be handled on a given server, resulting in greater utilization and lower hardware costs.

One technique for improving memory efficiency in virtualized systems is *content-based page sharing* (CBPS). In CBPS, duplicate blocks of memory (or 'pages') are collapsed into a single physical copy, with all duplicate virtual pages pointing back to the single physical page. This adjustment means that unneeded physical pages can be freed, lowering the memory footprint of the application or OS.

This form of memory deduplication has been experimented with in many virtualization platforms, and many of these systems have reported achieving impressive savings from sharing. For example, Difference Engine [6] reported absolute memory savings of 50% across three VMs (using page-level sharing), while VMware [19] reported savings as high as 40% across ten VMs.

These results are encouraging and appear to suggest that sharing can be used to great effect in real systems. However, there have been few studies of the real-world potential of page sharing across a variety of environments—i.e., in practice, how much sharing should we really expect to achieve? Other important issues have also received little scrutiny, such as the primary origin of sharing. Finally, there are some relevant questions about practices that may reduce sharing potential, such as randomizing or modifying memory contents for the sake of security. Practical investigation of these is-

sues is complicated by several factors, such as the need to tightly control experiments for comparison with other systems and the difficulty of gathering data from real-world machines. As a result, many systems have focused on comparing performance on benchmark workloads, which may not reflect real savings in practice.

In this paper, we conduct an empirical investigation of these questions about page sharing, under the assumption that all potential page sharing can be captured. This assumption is important because it separates the issue of evaluating a particular sharing system from evaluating the potential of page sharing itself. We conduct our investigation both on a set of memory traces captured from real-world users and on traces generated from a wide assortment of virtual machines.

Our major finding is that sharing tends to be significantly more modest (on the order of one half, or sometimes even less) than what might be expected from the literature. We also identify several interesting properties of page sharing, which both temper the expectations of sharing and may suggest alternative ways in which it should be deployed; these points are summarized below:

1. Self-sharing. We separate sharing into two essential categories: self-sharing (memory duplication within a single OS), and inter-VM sharing (memory duplication across multiple VMs). We find that in most cases, a significant majority (80% or more) of the sharing potential comes from self-sharing alone. This finding has important implications for the deployment of sharing systems. For instance, individual OS kernels may be nearly as well-suited to implementing sharing as conventional virtualized systems. Furthermore, such a shift would allow non-virtualized systems to take advantage of sharing.

2. Inter-VM sharing. Inter-VM sharing tends to be small, and effectively zero when the VMs are running heterogenous platforms. Sharing across VMs is more significant across a homogeneous platform, but even favorable conditions (e.g., cloned VMs) often derive a minority of savings (40% or less) from inter-VM sharing.

3. Sources of sharing. We provide a case study on the origins of sharing in a desktop environment by instrumenting the Linux kernel. Our system-aware memory tracing tool reveals how shared pages are used by programs – we find that the largest sharing source originates from GUI applications and related display libraries.

4. Security features and sharing. We find that OS security features, and address space layout randomization (ASLR) in particular, can reduce sharing by as much as 20%. Moreover, these features can reduce sharing across systems even when a high degree of homogeneity is imposed, such as in a virtual desktop environment.

In Section 2, we present background on sharing and motivation for why more careful study is needed. We detail our sharing model in Section 3 and describe our data and experimental results in Section 4. Our case studies on sharing sources and ASLR are presented in Sections 5 and 6, respectively. Finally, we mention related work in Section 7 and summarize our conclusions in Section 8.

2 Background and Motivation

Content-based page sharing has been targeted mostly at the hypervisor level within virtualized systems. In a virtualized system employing page sharing, the hypervisor (which has a global view of physical memory across all virtual machines) identifies sharable pages itself, then shares them without any input or cooperation from the virtual machine whose pages are being shared. In doing so, the primary aim is to capture sharing between VMs that are highly similar, significantly reducing the memory footprint of such VMs. This process is illustrated in Figure 1, in which six VM-level pages are serviced by only four physical pages.

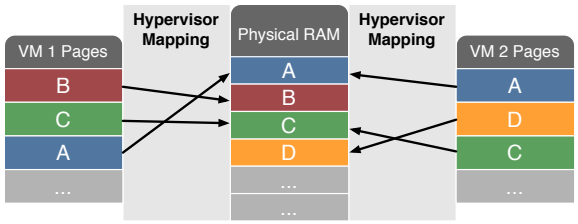


Figure 1: Page sharing between two VMs. The content of a page is indicated by its color/label.

Designing systems that can efficiently capture and exploit sharable pages has been the subject of a large volume of recent work. These efforts include both research systems ([6], [12], [17], [21]) as well as commercially available hypervisors ([2], [19]). Many of these systems report that 40% or more of a server's memory can be freed using page sharing techniques. These numbers are typically produced by grouping together several virtual machines on a host and reporting the total number of pages which could be freed due to sharing. However, this approach of simply tallying the total memory freed due to sharing does not offer any insights into how much memory is shared *internally to each VM* versus how much is shared *between VMs*.

During our investigations into improving page sharing in virtual environments, we uncovered a rarely discussed issue, namely that a large portion of the sharing found in virtualized environments is actually just internal to each VM. As a motivating example, we consider a simple benchmark used by other sharing systems ([6], [12]) – compiling the Linux kernel. Using two VMs booted from an identical Ubuntu 10.10 64-bit image, each with 1.5 GB memory, we compile the 2.6.32 kernel in each

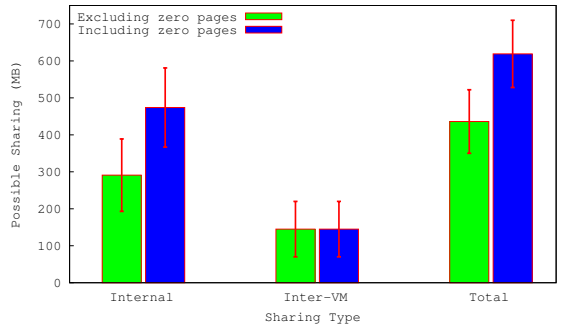


Figure 2: Sharing during kernel compilations in two identical Ubuntu VMs with and without zero pages.

(varying the minor version slightly to prevent caching of identical temporary files), and capture several memory snapshots spaced throughout each compile. We then consider each pair of snapshots across VMs and calculate the average internal sharing and average sharing across VMs. Figure 2 illustrates the total amount of sharing we found and the breakdown between internal "self-sharing" and inter-VM sharing. Excluding zero pages, we see that on average, 436 MB of the total memory could be shared, but of this, two-thirds (67%) was due to self-shared pages within individual VMs, with only the remaining third due to sharing between VMs. If we add in zero pages (which add only a single page to inter-VM sharing), the percentage of internal sharing rises to over three-quarters (77%).

This result runs counter to previous studies which have suggested that most page sharing is due to duplicated memory regions such as OS libraries that are common across multiple machines. Even in the previous example, in which the component VMs were identical and running nearly identical workloads, the majority of shared pages were sharable within isolated VMs. This has implications both for how memory sharing is managed, e.g., placing more VMs together may not significantly increase sharing levels, and how memory sharing is implemented, e.g., sharing within a single OS may be just as effective as attempting to do it across multiple VMs.

3 Types of Sharing

Let V be a virtual machine with a physical memory allotment m. For a given page size s, V has an array of m/s pages $P_V = \{p_1, p_2, \ldots, p_{m/s}\}$. Two pages p_i and p_j are *sharable* if their contents are byte-for-byte identical. For a given page p with k copies (including p), we can save $k-1$ pages by eliminating these duplicates and replacing them with references to the single copy p.

Let $UNIQUE(P_V)$ be the set of unique pages in P_V. Since this is the minimum number of pages needed to represent the memory of V, the **self-sharing** of V is the set of duplicate pages in P_V, given by

$$S_{self}(V) = P_V - UNIQUE(P_V)$$

Informally, self-sharing is simply the sharing that can be captured within a single virtual machine.

We can easily generalize this model to multiple VMs. Let G be the global set of VMs V_1 through V_k with corresponding page arrays P_{V_1} through P_{V_k}. The global page array is given by simply appending each component array: $P_G = \{P_{V_1}, P_{V_2}, \ldots, P_{V_k}\}$. We can then calculate $UNIQUE(P_G)$ and $S_{self}(G)$ as before. The **inter-VM sharing** of G is the total set of sharable pages in G *that cannot be captured by self-sharing alone*, given by

$$S_{inter}(G) = S_{self}(G) - \sum_{V \in G} S_{self}(V)$$

Informally, inter-VM sharing represents the sharing benefits that can be realized only by collocating the virtual machines in G. For a single machine V, $S_{inter}(V)$ is always zero, since all sharing is self-sharing. In general, if $S_{inter}(G) = 0$, then the VMs in G have no memory overlap—thus, no additional sharing can be captured by considering all VMs at once versus each VM in isolation.

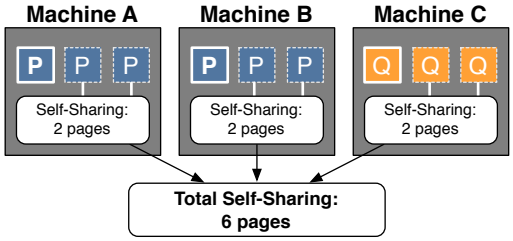


Figure 3: Sharing within single machines (self-sharing).

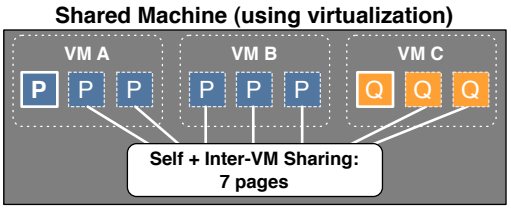


Figure 4: Inter-VM sharing between multiple machines.

We illustrate the difference between self-sharing and inter-VM sharing with a simple example, illustrated in Figures 3 and 4. Consider three machines A, B, and C that have sharable memory contents as shown in the figure: machines A and B both have three copies of page P and machine C has three copies of a different page Q. When we consider the machines individually, each can reduce its three pages down to one using self-sharing, for a total of six pages saved (two pages per machine). However, if A, B, and C are virtual machines residing on the

same physical host, as shown in Figure 4, the hypervisor may share a single copy of P globally for both VMs A and B—freeing one additional page compared to the self-sharing case. However, the shared page Q sees no benefit because the page is distinct only to VM C. In total, grouping the virtual machines together only results in an increase of one shared page which could not be found using self-sharing alone.

3.1 Sharing Properties

The model above illustrates how sharing can occur both within a single physical (or virtual) machine and between multiple virtual machines. Thus, we stress that while inter-VM sharing is only applicable to virtualized systems, self-sharing is just as possible in conventional systems. This distinction is important because most related work on sharing has focused heavily on virtual machines – this implicitly asserts that inter-VM sharing is the primary component of sharing.

Based on our sharing model, we can reason about when each type of sharing is likely to be more important. If individual machines have mostly unique pages, but there are many such machines (with some overlap), then hypervisor-level sharing – inter-VM sharing – is clearly favored. For example, if the memory of VM A is comprised of 10 distinct pages, and the memory of VM B is comprised of the same set of 10 pages, then inter-VM sharing can save 10 total pages (a 50% savings), while individually A and B could not achieve any savings via self-sharing. However, as the amount of self-sharing (duplicates within a single machine) rises, the relative importance of inter-VM sharing is likely to fall significantly. For example, if we take A and B and simply triple their memory contents (3 copies each of 10 distinct pages), then inter-VM sharing still provides the same 10 pages, but self-sharing provides 20 pages each, for a total of 40 pages – a 67% savings on top of which inter-VM sharing provides a modest 17% savings. In general, the potential of inter-VM sharing is bounded by the number of machines involved. This is because, for a given page p, inter-VM sharing can only save (at most) one page per machine. All other sharing involving p will be captured by self-sharing, and this amount of sharing is only bounded by the total number of pages available: k additional copies of p can add up to k shared pages (if they are all within the same machine). Inter-VM sharing is only likely to be more significant than self-sharing when there are many VMs with minimal internal redundancy.

3.2 Calculating Sharing

Our model is idealized in that real systems usually cannot actually capture *all* possible sharing. Some sharing opportunities may be short lived due to changing memory contents, and may either be missed by the system or passed over intentionally to reduce overhead. New systems are constantly attempting to identify both the largest and safest (long-lived) possible sharing opportunities, but we would like to sidestep this issue and simply tally all possibilities. Since proposing a deployable system is outside the scope of this work, we use a simple, heavyweight, but effective method of calculating sharing: scanning and snapshotting. Scanning refers to simply sequentially reading memory contents, hashing each page of memory, and outputting the list of hashes to be checked for duplicates. Since duplicate pages have duplicate hashes, checking for equality is trivial. Snapshotting refers to pausing the VM while scanning, which prevents memory from changing while looking for duplicates.

Scanning is used by some commercial systems such as VMware[19]—however, in order to prevent excessive overhead, scans can only be performed at moderately-spaced intervals (generally on the order of minutes), and may miss sharing due to changing memory. Since we are not concerned with efficiency, we simply suspend the VM, scan the memory snapshot, then resume. From the VM's perspective, this reduces the scan time to zero, so we can (in theory) perform scans as rapidly as desired. A second benefit of this approach is that since we read from the raw binary contents of memory, we can divide it into pages of any desired size.

One notable extension of the standard content-based sharing approach (proposed in [6]) is the use of small deltas between memory. This allows sharing pages even when they differ by a small amount, as opposed to restricting to when pages are exactly identical. We note that this type of sharing can be approximated by considering pages of smaller and smaller sizes, although decreasing the page size is only practical to a point, after which the overhead of handling an increasing number of pages becomes excessive.

Finally, our measurements of sharing explicitly ignore pages filled completely with zeroes. Zero pages are plentiful in many scenarios because the operating system and applications may zero out pages for future use; however, for the same reason, they are usually regarded as poor candidates for sharing. The intuition is straightforward – zero pages are plentiful but not likely to remain zero pages for long. Satori (being more concerned with sharing duration than most other projects) reported confirmation of this [12], and reported 20 times as much sharing from zero pages as from nonzero pages, but also reported that this page sharing was quite short lived. In our investigations, we adhere to the notion that zero pages are not a highly useful form of sharing, and disregard them in our experiments. This is important to note because including zero pages tends to greatly exaggerate results.

4 Evaluation

In this section, we present a study of sharing in actual machines and VMs. We pay particular attention to self-sharing versus inter-VM sharing in these systems, as well as exploring the effect of changing the underlying system (e.g., operating system version or application setup).

4.1 Data Collection

Our memory traces come from two sources – one set of traces captured from real-world machines, and a second set of traces generated from synthetic experiments. This both gives a holistic picture of real-world sharing potential and allows us to explore a wide set of systems.

Real-world memory traces. Our real-world memory traces come courtesy of the Memory Buddies project [21], which deployed a memory tracer onto a variety of UMass departmental machines and made these traces publicly available [11]. We focus on a set of traces gathered over a weeklong period from seven machines. The seven machines included four MacBooks (used by individual users) running similar versions of Mac OS X and three Linux servers running a variety of server tasks (web, mail, ssh, etc). The MacBooks each had 2 GB of RAM, and the Linux servers had 1, 4, and 8 GB of RAM. These are all production machines handling actual user workloads. Each machine produced a memory trace every thirty minutes during the time it was powered on (150 to 350 traces per machine), resulting in roughly 1700 traces in all. Each trace is comprised of one hash value per physical page (4 KB of memory).

Generated memory traces. To supplement our real-world traces, we configured a set of custom virtual machines. We produce a memory trace from a VM by suspending the VM, then reading the resulting binary memory image to create the hash list. This has the advantage (versus running a VM-based tracing tool) of having no impact on memory within the VM during tracing. Furthermore, since we can access the full binary contents of memory, we can use any sharing granularity desired. Sequential traces are produced by a series of suspend-resume cycles, and resetting the VM to a previous snapshot allows resetting memory to an exact previous state to evaluate the impact of a subsequent action or workload. We configured 10 distinct VMs, each with 1.5 GB physical memory:

- **Linux**: Ubuntu 10.04 and 10.10, and CentOS 5.3 (no GUI for CentOS). These distributions were chosen to be representative of typical desktop `apt` and server `rpm` based distributions. We consider both 32-bit and 64-bit versions (6 VMs total).
- **Windows**: Windows XP (x86) and Windows 7 x86 and x64 (3 VMs total).
- **Mac**: Mac OS X 10.6 Server (Snow Leopard). The server version is used due to virtualization restrictions, but is very similar to the desktop version.

For each of the 10 VMs, we use three application setups for capturing memory traces:

- **No applications**: The VM is freshly booted, but is not running any further software.
- **Server applications**: The VM runs a typical LAMP stack: Apache 2, PHP 5, and MySQL 5. The workload consists of issuing a series of MySQL queries and serving a mix of static and dynamic pages.
- **Desktop applications**: The VM runs a typical set of desktop applications: web browser (Firefox), office applications (OpenOffice), email client, media player, etc. The workload consists of opening several web pages and office documents (text document, spreadsheet, etc.) and playing a media file.

For each pair of VM and application setup, the VM is booted, the applications are loaded and the workload executed, then a memory snapshot is captured. This resulted in 28 traces total (no desktop traces for CentOS due to the lack of an installed GUI).

4.2 Sharing in Real-World Traces

We examined the real-world traces to determine if self-sharing comprised a substantial portion of total sharing. Since sharing can change substantially over time, we processed each set of sequential traces (for each of the seven host machines) and calculated the min, max, and average potential sharing over the weeklong recording period.

As discussed previously in Section 3.2, we exclude zero pages from all presented results, including both sharing percentages and absolute sharing. While we believe this decision more accurately reflects useful sharing, including zero pages in our results would have only a modest impact – for example, in nearly all cases, sharing from zero pages in our real-world traces comprises less than 5% of the total memory.

4.2.1 Self-Sharing

When we consider each machine individually, the sharing observed is entirely self-sharing. The amount of self-sharing in each machine is shown in Figure 5, both as a percentage of machine memory and as absolute sharing in MB. We see a modest, but not insignificant, level of self-sharing present in most of the machines – on average, about 14% of the total memory in any machine was sharable at any time. This demonstrates that a significant amount of duplicate memory is present even in isolated (non-virtualized) machines. Another interesting feature

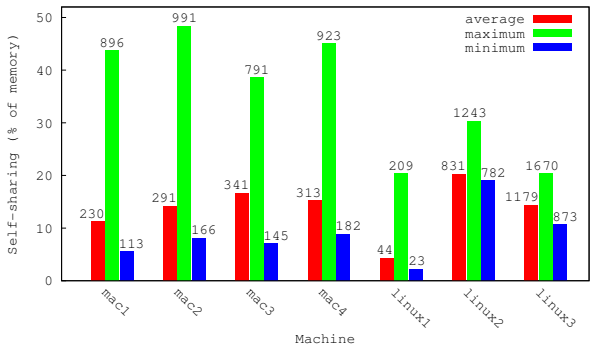


Figure 5: Self-sharing in the real-world memory traces, with absolute shares (in MB) noted.

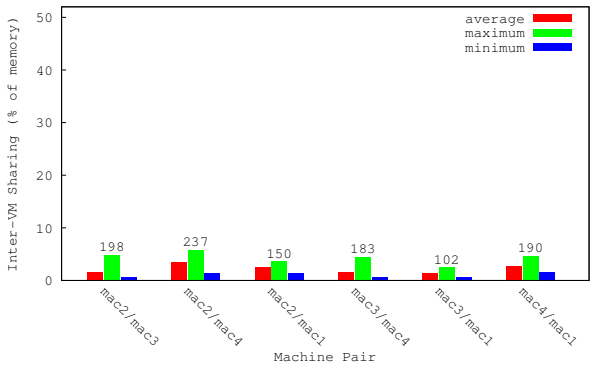


Figure 6: Inter-VM sharing between real-world machine pairs, with maximum absolute shares (in MB) noted.

is a difference of up to an order of magnitude between the minimum and maximum sharing observed, demonstrating the volatility of sharing over time. Since the average is much closer to the minimum than the maximum, this suggests brief periods of very high sharing potential surrounded by long periods of much lower sharing potential. Finally, we observe that the sharing variation is greater in the Macintosh machines than in the Linux machines – we speculate that this may be due to the fact that the desktop machines see a much wider range of applications compared to the servers which have a fixed purpose.

4.2.2 Inter-VM Sharing

Next, we consider the inter-VM sharing between pairs of machines represented in the real-world traces. For a given pair of machines (M_1, M_2), we wish to calculate both the self-sharing and inter-VM sharing between these machines. However, since there are hundreds of traces from each machine to select for (M_1, M_2), evaluating every possible trace pairing is infeasible. Thus, we simply randomly select several hundred pairs of traces from M_1 and M_2, then calculate the min, max, and average inter-VM sharing for (M_1, M_2) using this set. We perform this procedure for every possible (M_1, M_2) pair – since there are 7 machines, there are 21 total machine pairings.

The results for all Mac/Mac machine pairings are shown in Figure 6 – *for all other pairings*, including both Mac/Linux and Linux/Linux pairings, the inter-VM sharing observed was negligible (always under 1% of the total memory and usually under 0.1%). Even in the case of Mac/Mac pairings, as seen in Figure 6, the average inter-VM sharing is strikingly low, comprising only 2 to 3 percent of the total memory. Furthermore, even considering two traces selected for the greatest possible inter-VM sharing, this sharing never exceeded 6% of the total.

In the cases of greatest average inter-VM sharing, the machines involved showed average self-sharing in the 10 to 15 percent range. This means that even in the observed cases most favorable to inter-VM sharing, this sharing still only accounted for a mere 20% of the total sharing potential, with the other 80% attributable to self sharing. In the case of other pairings (which displayed effectively no inter-VM sharing), the memory gained from collocating only these machines would be negligible.

Considering more than 2 machines together naturally increases inter-sharing. However, we find the trends to be similar even when considering larger machine tuples. For example, if we consider traces from all 7 machines at once, we find a typical total sharing around 15% – however, *over 90%* of this sharing is solely attributable to self-sharing, with less than 10% being added by inter-VM sharing. In other words, even collocating all seven of these machines for the purpose of memory sharing would save very little – almost all possible sharing could be captured simply by having each machine eliminate duplicates within its own memory.

These results suggest that page sharing systems should not exclusively be directed at hypervisor-level systems such as Xen and VMware, but could also have worthwhile benefits when implemented in a commodity operating system such as Windows or Linux. Doing so would enable regular home users to reap the benefits of page sharing, and would allow virtual machines to maintain closer control over their memory usage. Moreover, we find that the added benefit from focusing sharing on virtualized systems may be substantially less than expected.

4.3 Platform Homogeneity

A set of VMs with a homogeneous underlying platform will have more sharing potential than a set of heterogenous machines, but the degree of homogeneity can vary widely. For example, a Windows 7 system may be quite different from an Ubuntu Linux system, but fairly similar to a Windows XP system. We can also consider factors such as architecture – for example, are Windows 7 x86 and Windows XP x86 closer or further apart than Windows 7 x86 and Windows 7 x64? As another exam-

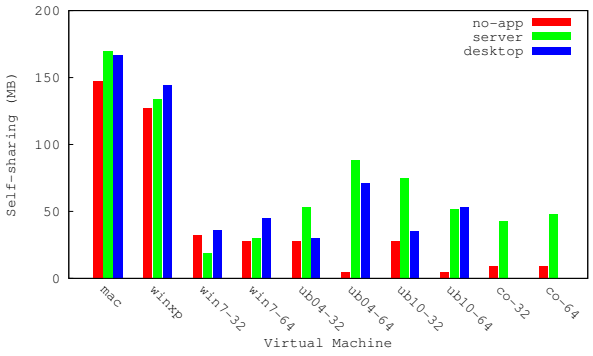


Figure 7: Self-sharing in the virtual machine traces.

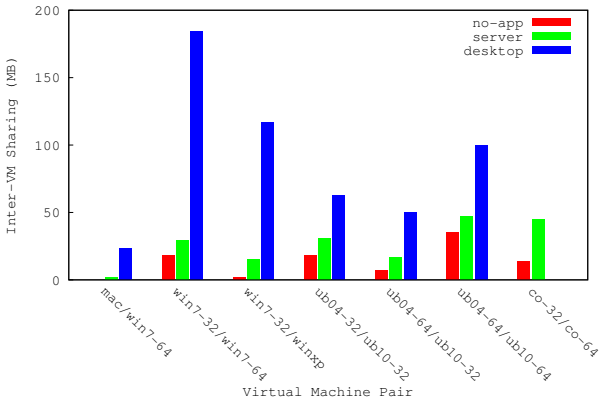


Figure 8: Inter-VM sharing between VM pairs.

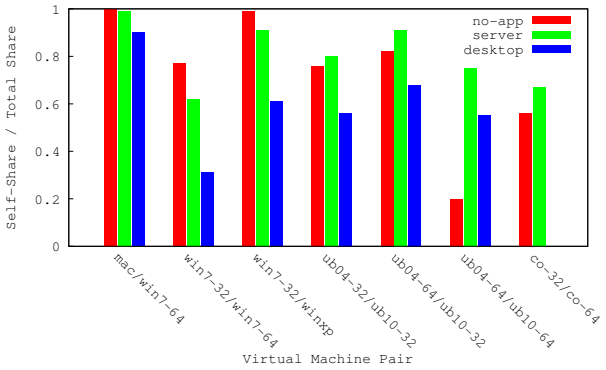


Figure 9: Self-sharing over total sharing among VM pairs.

ple, are the operating systems or user applications more important to compare with regards to sharing? We investigate these questions using our virtual machines and memory traces generated as detailed in Section 4.1.

4.3.1 Self-Sharing

As with our real-world traces, we first investigate the self-sharing present in our VMs. For each of the three application setups, the amount of self-sharing present in each VM is shown in Figure 7. The most notable feature seen is that the sharing in Mac OS X and Windows XP is much greater than in Windows 7 or any of the Linux VMs. This suggests that these systems tend towards a higher degree of internal memory redundancy, and may see greater benefit from harnessing this redundancy. Additionally, we see that in these cases, sharing does not substantially increase from the 'no-app' case when adding applications – this indicates that the base system is providing the bulk of the self-sharing, with only slight increases from the user-level applications.

Another interesting feature seen is that the base system sharing in both versions of Ubuntu decreases significantly when switching from a 32-bit system to a 64-bit system. Presumably, this indicates a lower level of redundancy in the 64-bit system libraries than in their 32-bit counterparts. Sharing in both versions of CentOS is quite low – we believe (based on observations discussed in Section 5) that this is largely due to the lack of a GUI.

4.3.2 Inter-VM Sharing

We next consider sharing between pairs of VMs. For every possible pairing, we calculate both the self-sharing and inter-VM sharing for each of the three application setups. While the complete results for all pairings are omitted for brevity, a representative sample is shown in Figures 8 and 9. Figure 8 shows the absolute inter-VM sharing for each of the selected pairings, while Figure 9 shows the relative importance of inter-VM sharing compared to self-sharing in each pairing.

Major OS mixes. Our first major observation is that inter-VM sharing is negligible in all mixes of major OSes (Mac OS X, Windows, or Linux). In all cases, these pairings closely resemble the `mac/win7-64` pair as seen in Figure 8 – there is essentially no inter-VM sharing in the base systems, and only a very small amount in the application setups. What little application sharing is present is likely due to shared resource files, hence why the sharing is significantly larger in the desktop case than the server case. Even with a common application setup, however, the inter-VM sharing represents only a minor fraction of the total sharing – less than 10%, with the rest attributable to self-sharing alone, as seen in Figure 9.

OS version mixes. Our second observation is that with different versions of a common major platform (e.g., Ubuntu 10.04 vs 10.10), sharing is significantly reduced in the base system, but is not strongly reduced in the case of common applications. The best example of this is seen in the `win7-32/winxp` pairing – although there is almost no inter-VM sharing in the base system ($< 5\%$), the amount of inter-VM sharing increases significantly in the application setups (roughly 40% inter-VM sharing in the desktop case). This is due to the fact that many of the applications themselves (e.g., OpenOffice)

run the same version on these systems, adding a significant amount of sharing. This behavior is also seen in the various Ubuntu pairings – mixing versions significantly degrades inter-VM sharing in the base system (e.g., 75% self-sharing between Ubuntu 10.04 and 10.10 32-bit), but these cases still show significant application sharing.

Architecture mixes. Overall, we see similar behavior when changing OS architecture (32-bit to 64-bit or vice versa) as when changing the OS version. However, changing the architecture is still significantly less disruptive than changing the major OS version, which completely eliminates most sharing.

The one notable case in which we see an architecture-specific behavior is when pairing the two 64-bit Ubuntu versions (`ub04-64/ub10-64`). In this case, almost all sharing in the base systems is due to inter-VM sharing (80%) rather than self-sharing – this is likely due to the fact that both of these systems displayed minimal self-sharing themselves, as seen previously in Figure 7.

Application types. In all cases (except CentOS, which did not run a GUI), we see that sharing was substantially higher in the GUI desktop applications than in the server applications. This may be partially due simply to the higher memory footprint of our desktop applications, but is also likely due to the tendency of GUI-related libraries to increase memory redundancy. We explore this tendency in Section 5.

4.4 Variable-Sized Hashing

Sharing is typically done on a page-by-page basis (that is, only sharing at the granularity of an entire page). However, one can also share on a different granularity, trading off between sharing potential and efficiency – a smaller granularity increases overhead, but is capable of sharing smaller chunks of memory. Since operating systems allocate memory on a per-page basis, it is most natural to consider even multiples (0.5, 2, 4) of the base 4 KB page size. Again, however, there is no requirement to share using these granularities. Thus, we examined several of our traces with sharing granularities varying from 0.4 to 2.4 in intervals of 0.1. The results for a typical trace (taken from an Ubuntu VM) are shown in Figure 10.

As expected, sharing increases modestly as the granularity increases. We also note the significant peaks at 0.5, 1, and 2 hashes per page (the evenly-dividing settings). The greatest ratio of sharable memory to hashes per page (a proxy for processing overhead) is still at the standard 1-page granularity. This is in line with other reported results [21] that have suggested modest but diminishing returns from increasing the sharing granularity. Furthermore, these results confirm that the only reasonable granularities evenly divide the page size, as other granularities significantly reduce possible sharing.

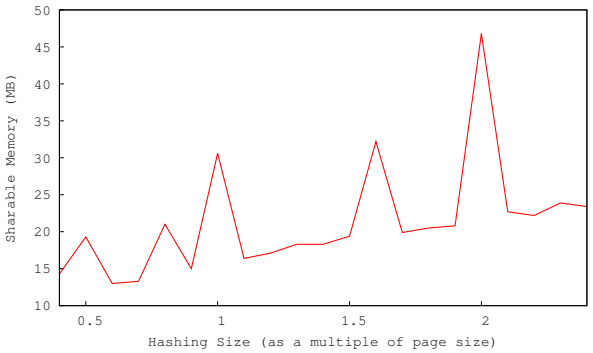


Figure 10: Self-sharing with variable hashing sizes.

5 Sources of Self-Sharing

Our previous results demonstrate the importance of self-sharing, but do not explain where this self-sharing originates. The root of self-sharing – internal redundancy – presumably only exists by accident. However, we have seen that that redundancy is common in all systems studied. To shed light on what causes redundancy, we have conducted a case study on Linux desktop applications.

5.1 An Extended Memory Tracer

Since identifying the source of sharing requires more information than basic memory traces, we extended the memory tracer (a kernel module) used to collect our real-world Linux traces. The original tracer simply walks through each page of memory and calculates a hash based on the page contents. For pages in use by active processes, our extended tracer also collects two additional pieces of information:

- The content type of the page – either a specific part of a regular program address space (e.g., stack, heap), or a mapped page of a shared library.
- The process(es) using the page. For a shared library page, there may be any number of processes using the page. For other pages, there will only be one process using the page.

For example, two pages might give the following (omitting the memory content hash values):

```
[libc-2.12.so 000b6000 r-xp]: sshd apache2
[heap]: mysqld
```

The first is a specific page of `libc`, in use by SSH and Apache. The second is a page in the MySQL heap. Using this extended information, we can calculate not only the amount of sharing possible, but which processes or libraries are actually involved in the sharing.

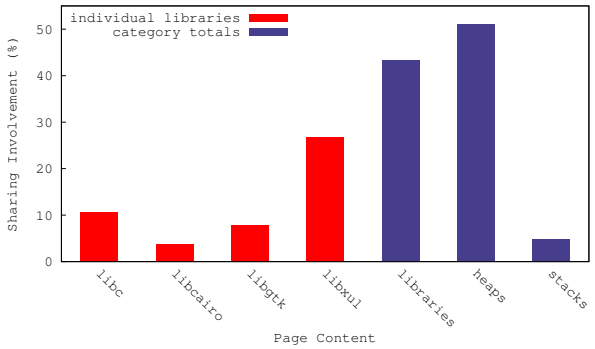


Figure 11: Sharing by content as a percentage of total sharing.

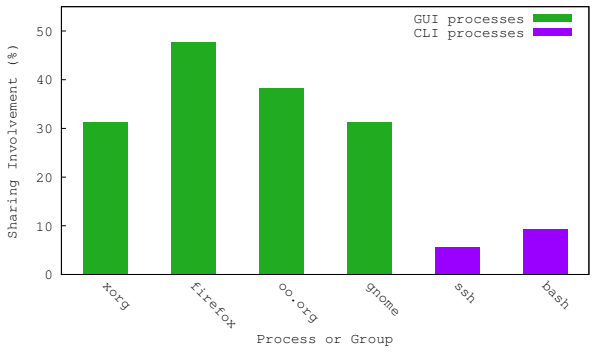


Figure 12: Sharing by process as a percentage of total sharing.

5.2 Case Study: Desktop Applications

A growing trend in virtualized desktop systems is the use of thin clients that communicate with homogeneous server-side VMs. To explore this potentially lucrative source of sharing, we conduct a case study of desktop applications in an Ubuntu VM. We examine sharing opportunities by page type, then discuss some of the main applications which exhibit high levels of sharing.

The primary page types we examine are stack pages, heap pages, and shared library pages. The results for each of these, as well as for a few individual shared libraries, are shown in Figure 11 (excluding zero pages as before). Note that these results are not additive, since multiple page types (e.g., a stack page and a heap page) may have the same content, which causes both to be included in the same chunk of sharing.

We see that the largest single source of sharing is heap pages, which are involved in over 50% of all sharing within the VM. Library pages are close behind, involved in 43% of all sharing. Stack pages, in contrast, are relatively minor, at less than 5% involvement. Of the shared libraries, the single library involved in the most sharing is `libxul`, the user interface library used by Firefox. Other libraries, such as `libc`, are used more widely, but were involved in substantially less sharing overall.

We also looked at sharing involvement by process rather than content. Selected processes are shown in Figure 12 (note that these results are again not additive, since multiple processes may be using a single page). The single most important process to sharing was Firefox, which was involved in nearly half of all sharing – this is understandable given the importance of `libxul` to sharing seen in Figure 11. However, we also see substantial sharing in other GUI applications such as OpenOffice (38%) as well as system-wide GUI processes such as the X server (31%) and all Gnome-related processes (also 31%). In contrast, headless applications such as SSH (6%) were involved in much less sharing.

The importance of GUI applications to sharing is further reinforced when we examine the most widely shared individual pages (i.e., those pages with the most copies). Every one of the top six shared pages, which alone are responsible for almost 10 MB of sharing, are used by Firefox, Xorg, or both. The single most shared page, with 597 distinct copies (2.3 MB shared) was a heap page used by Xorg. These situations are likely byproducts of repeatedly allocating and freeing a particular data structure, since copies may include old pages that have yet to be reclaimed by the OS as well as pages currently in use.

6 Memory Security and Sharing

Operating system designers are constantly seeking new ways to harden their systems against attackers and reduce exploit opportunities. One common attack vector involves overwriting memory contents (e.g., a buffer overflow attack), particularly when the overwritten memory is at a known address, such as a key library function. To harden systems against these kind of attacks, modern operating systems employ a technique known as **address space layout randomization**, or ASLR, in which the location of key libraries and program assets (e.g., the heap) in memory are randomized. This randomization is designed to prevent attackers from making use of known addresses (e.g., in corrupting targeted data). For the example address space shown in Figure 13a, two possible randomizations are shown in Figures 13b and 13c.

While ASLR is exclusively a security measure, since it modifies the contents of memory, it has the potential to affect the level of page sharing possible. Two VMs that may be running identical software may have different memory contents if ASLR is enabled, resulting in less possible sharing. For example, the randomized address spaces shown in Figures 13b and 13c may share less than if they were not randomized (and hence equal to Figure 13a). We conduct a study on the impact of ASLR on sharing potential by evaluating multiple implementations across several operating systems.

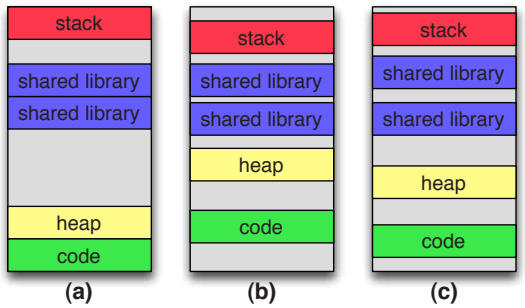


Figure 13: A non-randomized address space (a) and two examples of the address space after randomization (b and c).

6.1 Current ASLR Support

While ASLR adoption has been gradual and the level of support varies among operating systems, most popular OSes have at least rudimentary support and are moving towards more complete instrumentation. Most implementations allow for enabling or disabling randomization, which we exercise in studying its impact. We selected four implementations for study – two in Linux, one in Windows 7, and one in Mac OS X Lion.

Linux. The Linux kernel first introduced ASLR support in 2005, and modern versions randomize the major components of a process (library locations, stack, heap, code) [8]. The ability to toggle system-wide ASLR is provided via the `/proc` interface, as well as an intermediate setting in which heap randomization is not used, but all other randomizations are.

PaX. An alternate implementation of ASLR for Linux is provided by PaX [13], which is a patch for the mainline Linux kernel aimed at improving overall security. A PaX-enabled kernel is used by several 'hardened' distributions of Linux aimed at maximizing security, and can also be deployed in most normal distributions. The ASLR implementation in PaX provides several features not provided by the standard Linux implementation, such as randomization within the kernel itself [7].

Windows. Microsoft introduced ASLR support in Windows Vista and continued in Windows 7, providing randomization of the stack, heap, DLLs, and so forth [20]. ASLR is enabled on a per-application basis, and is opt-in by default. While most system-provided applications within Windows enable ASLR, third-party application support has been slow [16]. However, Microsoft recently released a utility [5] that provides the ability to forcibly enable or disable ASLR for particular processes. In our tests, we encountered no ill effects from enabling it for applications that do not opt-in by default.

Mac OS X. Apple first introduced a simplistic form of ASLR in Mac OS X 10.6 (Snow Leopard), and support was expanded in 10.7 (Lion) [9]. Unfortunately, there is presently no straightforward way to disable randomization within Lion – the only known method [10] relies on setting a particular POSIX flag during process creation. To leverage this, we write a script that simply sets this flag, then spawns the target application (which runs without randomization).

6.2 Evaluating ASLR's Sharing Impact

For each of the four ASLR implementations, we wish both to identify whether randomization has an impact on sharing, and if so, to determine the extent of this impact. To do this, we simulate a scenario in which many VMs are booted from an identical base image. This is a lucrative scenario both for virtualization and for page sharing, and represents a case in which users are likely to care about fine-tuning sharing potential. We run this scenario for each of the three host operating systems – Ubuntu 10.10 64-bit (for both the standard Ubuntu kernel and a patched PaX kernel, both version 2.6.32), Windows 7 64-bit, and Mac OS X 10.7.

For a single OS, our test procedure (using a single VM) is as follows. First, we globally disable ASLR, using one of the tools mentioned in the previous section (note that in the case of Mac OS X, we cannot disable system randomization). We then reboot the VM to reset memory to a reliable state. Then, we populate the VM's memory by opening a predefined list of applications (web browser, text editor, office software, music player, etc.) using a shell script or batch file. After letting the contents of memory settle, we capture this 'non-randomized' memory snapshot. We then globally enable ASLR, reboot, and then repeat the snapshot procedure again to obtain a 'randomized' snapshot.

To simulate booting multiple VMs from the same image, we repeat this four times, resulting in a set of four randomized snapshots and a set of four nonrandomized snapshots. Since the only substantive difference between the sets is whether randomization is used, any significant reduction in sharing in the randomized snapshot set should be due to ASLR – furthermore, the use of multiple snapshots averages any other memory differences that occur between reboots.

The results, as a percentage reduction in sharing, are shown in Figure 14. The total sharing reduction is further broken down into self-sharing and inter-VM sharing – note that these are not additive, since they do not contribute equally to the total sharing. We see a modest, but noticeable reduction in sharing across all implementations. The largest reduction is seen in Windows 7, in which total sharing was reduced by 13% (in line with [18], which reported a 16% reduction in Windows 7). Total sharing in Mac OS X was reduced by only 3% – however, as noted above, randomization was not disabled for the system itself, and hence this result is conservative.

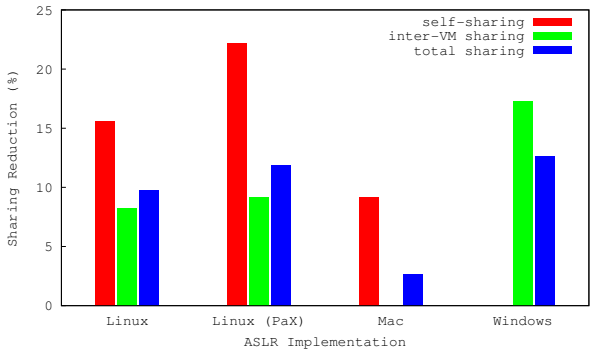


Figure 14: Reduction in page sharing observed with a variety of ASLR implementations.

Overall sharing in Linux was reduced by 10% using the stock kernel, while the PaX-patched kernel reduced sharing by a slightly higher 12%.

The results of the sharing breakdown are somewhat more surprising. In the case of Linux, ASLR reduced sharing both within single systems and across systems. However, in the case of Windows, the entire reduction was due to inter-VM sharing – individual systems shared the same amount regardless of randomization. This suggests that in a non-virtualized environment, sharing in Windows is not impacted at all by ASLR. Oddly, we observe the opposite behavior in Mac OS X – here, all reductions were due to self-sharing, with no reduction observed in sharing across systems. One reason for this may be the relative amount of self-sharing – as we observed in Section 4.3.1, self-sharing in Mac OS X was much higher than in Windows 7.

7 Related Work

One of the first systems implementing page sharing was the Disco virtual machine monitor (VMM) [3], which proposed sharing identical code pages (typically read-only) between virtual machines, as well using as a shared copy-on-write disk to share disk contents between VMs. For sharing memory, the original technique of content-based page sharing was introduced in VMware ESX Server [19]. CBPS has received significant attention because it requires no assistance from the guest operating system, and thus can be performed transparently within the virtualization layer.

The original virtual-machine centric nature of most page sharing work has persisted past the original VMware and Disco papers, and provides an important motivational basis of most page sharing systems today. One possible reason for this (besides simple precedent set by earlier papers) is that it seems intuitive that groups of virtualized systems could provide significant gains that cannot be realized within a single operating system. In contrast to this focus, we argue that page sharing need not be targeted exclusively at virtualized environments.

Several systems have built on the core CBPS idea to further exploit potential savings. The Potemkin VMM [17], which was originally built on Xen, makes heavy use of page sharing by creating new virtual machines as clones of an existing VM image. The clones are then able to store only differences from the base image rather than allocating their own dedicated memory space, in effect replacing a memory partition with a single base image and a set of memory deltas. This is an example of a special type of virtualized environment which is undoubtedly well suited to page sharing, since a new VM is (by definition) initially able to share 100% of its pages with the base VM. We explore several setups of this type and find that it is the exception to our general observation that self-sharing from individual virtual machines comprises the vast majority of sharing in groups of machines.

Another extension of the base CBPS technique has been sub-page sharing, in which full memory pages are broken down into pieces to allow for finer grained sharing. This technique was applied both in Difference Engine [6] (based on Xen) and Memory Buddies [21] (based on ESX Server). Difference Engine also went a step further and considered storing small 'patches' between memory, rather than just uniform sub-page sharing. Additionally, Difference Engine employed compression of non-shared VM memory to increase overall memory efficiency. However, they reported that the majority of sharing benefits were attributable to the basic page sharing paradigm rather than additional enhancements. Intelligent collocation of multiple VMs to optimizing sharing was explored using memory fingerprints in [21] and hierarchical tree models in [15].

A system for sharing memory in massively parallel applications using a distributed hash table was proposed in [22]. This work also considers the distinction between inter-node and intra-node sharing in parallel applications (in which inter-VM sharing is relatively more significant). We complement this work by focusing on sharing in general purpose, single-node virtualized systems.

The Satori [12] system took a different approach to sharing altogether and argued that most potential sharing lasts only a few seconds rather than at least a few minutes. A typical CBPS approach involving periodically scanning memory is insufficient for capturing this type of sharing, since scans cannot be performed on the granularity of a few seconds for performance reasons. Satori implemented sharing by watching for identical regions of memory when read from disk – this is also the approach that was integrated into Xen. However, the downside to this approach is that it requires modifications to the guest operating systems themselves. This represents a trade off between transparent sharing requiring guest modifi-

cations (the Satori/Xen approach) and oblivious sharing requiring no such modifications (the VMware approach).

Recent versions of the Linux kernel include sharing functionality via the KSM kernel module [1], which uses a scanning approach similar to that of VMware. While originally developed for sharing within the Linux KVM virtualization infrastructure, KSM can also be used for sharing within a nonvirtualized Linux system. An evaluation of KSM in a virtualized setting is given in [4]. The Singleton system [14], also based on KSM, focused on optimizing caching for low overhead deduplication.

8 Conclusion

Page sharing presents an opportunity to increase memory efficiency, but understanding the dynamics affecting page sharing are important to maximizing its benefits. We present an investigation and analysis into the sources of page sharing, and find that in many cases, the majority of sharing potential is attributable to redundancy within single machines (self-sharing) rather than between multiple machines (inter-VM sharing). This suggests (1) that sharing may be effectively exploited at the level of a single VM rather than a hypervisor, and (2) that sharing need not be restricted to virtualized systems at all.

For sharing across VMs, we investigate several application platforms and find that operating system homogeneity is the most important component of inter-VM sharing, with application, architecture, and version homogeneity of lesser (but still significant) importance. In particular, we see effectively no sharing between different base platforms – e.g., between a Windows and Linux system. Inter-VM sharing is still present, but greatly reduced, by changing the version of the base system.

We also conduct a case study of self-sharing within the Linux kernel and find that GUI applications and associated system libraries are the clearest source of self-sharing potential. We leave porting our tracing tool to non-Linux systems as future work to investigate whether similar trends will hold in other operating systems. Finally, we explore the impact of address space layout randomization on sharing potential. We find that in all major systems, this feature has a measurable negative impact on sharing potential. We are continuing to explore the factors behind both favorable and unfavorable sharing scenarios, and believe that understanding these issues will enable more efficient memory usage in both virtualized and nonvirtualized systems.

Acknowledgements. We would like thank our shepherd, Haibo Chen, and the anonymous reviewers for their valuable comments and feedback. This work is supported by NSF grants CNS-0519894, CNS-1117221, CNS-1143655, CNS-0916972, and CNS-0855128.

References

[1] ARCANGELI, A., EIDUS, I., AND WRIGHT, C. Increasing memory density by using ksm. In *OLS* (July 2009).

[2] 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 *SOSP* (Oct. 2003).

[3] BUGNION, E., DEVINE, S., AND ROSENBLUM, M. Disco: Running commodity operating systems on scalable multiprocessors. In *TOCS* (Nov. 1997).

[4] CHANG, C.-R., WU, J.-J., AND LIU, P. An empirical study on memory sharing of virtual machines for server consolidation. In *ISPA* (May 2011).

[5] The enhanced mitigation experience toolkit. `http://support.microsoft.com/kb/2458544`, 2011.

[6] 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. In *OSDI* (Dec. 2008).

[7] Kernel hardening roadmap. `https://wiki.ubuntu.com/SecurityTeam/Roadmap/KernelHardening`, 2012.

[8] Linux kernel aslr implementation. `http://xorl.wordpress.com/2011/01/16/linux-kernel-aslr-implementation/`, 2011.

[9] Mac os x lion security. `http://www.apple.com/macosx/whats-new/features.html#security`, 2011.

[10] How gdb disables aslr in mac os x lion. `http://reverse.put.as/2011/08/11/how-gdb-disables-aslr-in-mac-os-x-lion/`.

[11] Memory buddy trace repository. `http://traces.cs.umass.edu/index.php/CpuMem/CpuMem`, 2009.

[12] MIŁÓS, G., MURRAY, D. G., HAND, S., AND FETTERMAN, M. A. Satori: enlightened page sharing. In *USENIX ATC* (June 2009).

[13] Pax. `http://pax.grsecurity.net/`, 2012.

[14] SHARMA, P., AND KULKARNI, P. Singleton: System-wide page deduplication in virtual environments. In *HPDC* (June 2012).

[15] SINDELAR, M., SITARAMAN, R. K., AND SHENOY, P. Sharing-aware algorithms for virtual machine colocation. In *SPAA* (June 2011).

[16] Dep/aslr implementation progress in popular third-party windows applications. White paper, Secunia Research, 2010.

[17] VRABLE, M., MA, J., CHEN, J., MOORE, D., VANDEKIEFT, E., SNOEREN, A. C., VOELKER, G. M., AND SAVAGE, S. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *SOSP* (Oct. 2005).

[18] Project vrc (virtual reality check). `http://www.projectvrc.com/white-papers`, 2010.

[19] WALDSPURGER, C. A. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev. 36* (Dec. 2002).

[20] WHITHOUSE, O. An analysis of address space layout randomization on windows vista. White paper, Symantec Advanced Threat Research, `http://www.symantec.com/avcenter/reference/Address_Space_Layout_Randomization.pdf`, 2007.

[21] WOOD, T., TARASUK-LEVIN, G., SHENOY, P., DESNOYERS, P., CECCHET, E., AND CORNER, M. D. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. In *VEE* (Mar. 2009).

[22] XIA, L., AND DINDA, P. A case for tracking and exploiting inter-node and intra-node memory content sharing in virtualized large-scale parallel systems. In *VTDC* (June 2012).

Primary Data Deduplication – Large Scale Study and System Design

Ahmed El-Shimi Ran Kalach Ankit Kumar Adi Oltean Jin Li Sudipta Sengupta
Microsoft Corporation, Redmond, WA, USA

Abstract

We present a large scale study of primary data deduplication and use the findings to drive the design of a new primary data deduplication system implemented in the Windows Server 2012 operating system. File data was analyzed from 15 globally distributed file servers hosting data for over 2000 users in a large multinational corporation.

The findings are used to arrive at a chunking and compression approach which maximizes deduplication savings while minimizing the generated metadata and producing a uniform chunk size distribution. Scaling of deduplication processing with data size is achieved using a RAM frugal chunk hash index and data partitioning – so that memory, CPU, and disk seek resources remain available to fulfill the primary workload of serving IO.

We present the architecture of a new primary data deduplication system and evaluate the deduplication performance and chunking aspects of the system.

1 Introduction

Rapid growth in data and associated costs has motivated the need to optimize storage and transfer of data. Deduplication has proven a highly effective technology in eliminating redundancy in backup data. Deduplication's next challenge is its application to primary data - data which is created, accessed, and changed by end-users, like user documents, file shares, or collaboration data. Deduplication is challenging in that it requires computationally costly processing and segmentation of data into small multi-kilobyte chunks. The result is large metadata which needs to be indexed for efficient lookups adding memory and throughput constraints.

Primary Data Deduplication Challenges. When applied to primary data, deduplication has additional challenges. First, expectation of high or even duplication ratios no longer hold as data composition and growth is not driven by a regular backup cycle. Second, access to data is driven by a constant primary workload, thus heightening the impact of any degradation in data access performance due to metadata processing or on-disk fragmentation resulting from deduplication. Third, deduplication must limit its use of system resources such that it does not impact the performance or scalability of the primary workload running on the system. This is paramount when applying deduplication on a broad platform where a variety of workloads and services compete for system resources and no assumptions of dedicated hardware can be made.

Our Contributions. We present a large and diverse study of primary data duplication in file-based server data. Our findings are as follows: (i) Sub-file deduplication is significantly more effective than whole-file deduplication, (ii) High deduplication savings previously obtained using small ∼4KB variable length chunking are achievable with 16-20x larger chunks, after chunk compression is included, (iii) Chunk compressibility distribution is skewed with the majority of the benefit of compression coming from a minority of the data chunks, and (iv) Primary datasets are amenable to partitioned deduplication with comparable space savings and the partitions can be easily derived from native file metadata within the dataset. Conversely, cross-server deduplication across multiple datasets surprisingly yields minor additional gains.
Finally, we present and evaluate salient aspects of a new primary data deduplication system implemented in the Windows Server 2012 operating system. We focus on addressing the challenge of scaling deduplication processing resource usage with data size such that memory, CPU, and disk seek resources remain available to fulfill the primary workload of serving IO. The design aspects of our system related to primary data serving (beyond a brief overview) have been left out to meet the paper length requirements. The detailed presentation and evaluation of those aspects is planned as a future paper.

Paper Organization. The rest of this paper is structured as follows. Section 2 provides background on deduplication and related work. Section 3 details data collection, analysis and findings. Section 4 provides an overview of the system architecture and covers deduplication processing aspects of our system involving data chunking, chunk indexing, and data partitioning. Section 5 highlights key performance evaluations and results involving these three aspects. We summarize in Section 6.

2 Background and Related Work

Duplication of data hosted on servers occurs for various reasons. For example, files are copied and potentially modified by one or more users resulting in multiple fully or partially identical files. Different documents may

share embedded content (e.g., an image) or multiple virtual disk files may exist each hosting identical OS and application files. Deduplication works by identifying such redundancy and transparently eliminating it.

Related Work. *Primary* data deduplication has received recent interest in storage research and industry. We review related work in the area of data deduplication, most of which was done in the context of *backup* data deduplication.

Data chunking: Deduplication systems differ in the granularity at which they detect duplicate data. Microsoft Storage Server [5] and EMC's Centera [12] use file level duplication, LBFS [18] uses variable-sized data chunks obtained using Rabin fingerprinting [22], and Venti [21] uses individual fixed size disk blocks. Among content-dependent data chunking methods, Two-Threshold Two-Divisor (TTTD) [13] and bimodal chunking algorithm [14] produce variable-sized chunks. *Winnowing* [23] has been used as a document fingerprinting technique for identifying copying within large sets of documents.

Backup data deduplication: Zhu et al. [26] describe an inline backup data deduplication system. They use two techniques to reduce lookups on the disk-based chunk index. First, a bloom filter [8] is used to track existing chunks in the system so that disk lookups are avoided on non-existing chunks. Second, portions of the disk-based chunk index are prefetched into RAM to exploit sequential predictability of chunk lookups across successive backup streams. Lillibridge et al. [16] use *sparse indexing* to reduce in-memory index size at the cost of sacrificing deduplication quality. HYDRAstor [11] is a distributed backup storage system which is content-addressable and implements global data deduplication, and serves as a back-end for NEC's primary data storage solutions.

Primary data deduplication: DEDE [9] is a decentralized host-driven block-level deduplication system designed for SAN clustered file systems for VMware ESX Server. Sun's ZFS [6] and Linux SDFS [1] provide inline block-level deduplication. NetApp's solution [2] is also at the block level, with both inline and post-processing options. Ocarina [3] and Permabit [4] solutions use variable size data chunking and provide inline and post-processing deduplication options. iDedup [24] is a recently published primary data deduplication system that uses inline deduplication and trades off capacity savings (by as much as 30%) for performance.

3 Data Collection, Analysis, and Findings

We begin with a description of datasets collected and analysis methodology and then move on to key findings.

Workload	Srvrs	Users	Total Data	Locations
Home Folders (HF)	8	1867	2.4TB	US, Dublin, Amsterdam, Japan
Group File Shares (GFS)	3	*	3TB	US, Japan
Sharepoint	1	500	288GB	US
Software Deployment Shares (SDS)	1	†	399GB	US
Virtualization Libraries (VL)	2	†	791GB	US
Total	15		6.8TB	

Table 1: Datasets used for deduplication analysis. *Number of authors (users) assumed in 100s but not quantifiable due to delegated write access. †Number of (authors) users limited to < 10 server administrators.

3.1 Methodology

We selected 15 globally distributed servers in a large multinational corporation. Servers were selected to reflect the following variety of file-based workload data seen in the enterprise:

- Home Folder servers host the file contents of user home folders (Documents, Photos, Music, etc.) of multiple individual users. Each file in this workload is typically created, modified, and accessed by a single user.
- Group File Shares host a variety of shared files used within workgroups of users. Each file in this workload is typically created and modified by a single user but accessed by many users.
- Sharepoint Servers host collaboration office document content within workgroups. Each file in this workload is typically modified and accessed by many users.
- Software Deployment shares host OS and application deployment binaries or packed container files or installer files containing such binaries. Each file in this workload is typically created once by an administrator and accessed by many users.
- Virtualization Libraries are file shares containing virtualization image files used for provisioning of virtual machines to hypervisor hosts. Each file in this workload is typically created and updated by an administrator and accessed by many users.

Table 1 outlines the number of servers, users, location, and total data size for each studied workload. The count

Dataset	Dedup Space Savings		
	File Level	Chunk Level	Gap
VL	0.0%	92.0%	∞
GFS-Japan-1	2.6%	41.1%	15.8x
GFS-Japan-2	13.7%	39.1%	2.9x
HF-Amsterdam	1.9%	15.2%	8x
HF-Dublin	6.7%	16.8%	2.5x
HF-Japan	4.0%	19.6%	4.9x
GFS-US	15.9%	36.7%	2.3x
Sharepoint	3.1%	43.8%	14.1x

Table 2: Improvement in space savings with chunk-level dedup vs. file-level dedup as a fraction of the original dataset size. Effect of chunk compression is *not* included. (The average chunk size is 64KB.)

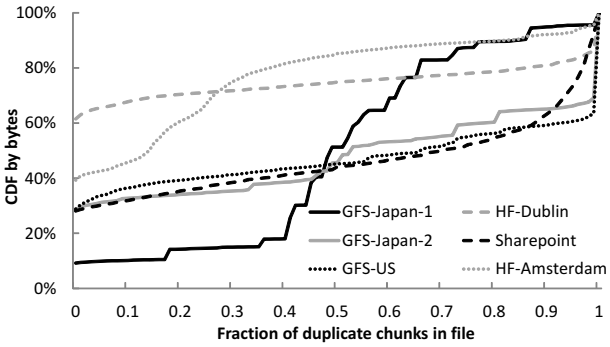


Figure 1: CDF of original (un-deduplicated) files by bytes, as a function of the fraction of chunks in a file that are duplicate, for various datasets.

of users reflects the number of content authors, rather than consumers, on the server.

Files on each server were chunked using a Rabin fingerprint [22] based variable sized chunker at different chunk sizes and each chunk was hashed using a SHA-1 [19] hash function. Each chunk was then compressed using gzip compression [25]. The resulting chunk size (before and after compression), chunk offset, SHA-1 hash, and file information for each chunk were logged and imported into a database for analysis.

To allow for detailed analysis of the effects of different chunking and compression techniques on specific file types, file group specific datasets where extracted from the largest dataset (GFS-US) to represent Audio-Video, Images, Office-2003, Office-2007, PDF and VHD (Virtualization) file types.

3.2 Key Findings

Whole-file vs. sub-file dedup: In Table 2, we compare whole-file and sub-file chunk-level deduplication. Whole-file deduplication has been considered earlier for primary data in the form of *single instance storage* [7]. The study in Meyer et al. [17], involving full desktop

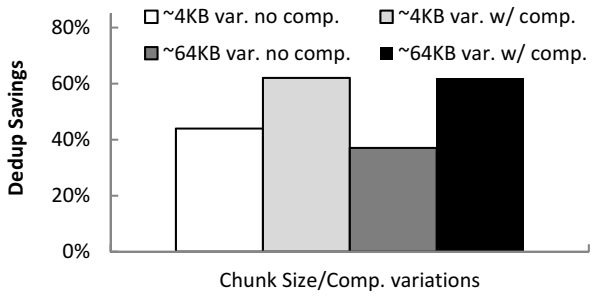


Figure 2: Dedup space savings (%) for chunking with ∼4KB/∼64KB variable chunk sizes, with impact of compression, for GFS-US dataset.

file systems in an enterprise setting, found that file-level deduplication achieves 75% of the savings of chunk-level deduplication. Table 2 shows a significant difference in space savings for sub-file dedup over whole-file dedup, ranging from 2.3x to 15.8x to ∞ (the latter in cases where whole file dedup gives no space savings). The difference in our results from Meyer's [17] is likely due to the inclusion of OS and application binary files which exhibit high whole file duplication in the earlier study [17] while this study focuses on user authored files which are more likely to exhibit sub-file duplication.

Figure 1 provides further insight into the big gap in space savings between chunk-level dedup and file-level dedup for several datasets. On the x-axis, we sort the files in increasing order of fraction of chunks that are duplicate (i.e., also occur in some other file). On the y-axis, we plot the CDF of files by bytes. We calculate a CDF of deduplication savings contributed by files sorted as in this graph, and find that the bulk of the duplicate bytes lie between files having 40% to 95% of duplicate chunks. Hence, sub-file dedup is necessary to identify duplicate data that occurs at a granularity smaller than that of whole files.

We also found reduced space savings for chunking with 64KB fixed size blocks when comparing it with ∼64KB variable size chunking (chunk compression included in both cases). This is because chunking with fixed size blocks does not support the identification of duplicate data when its duplicate occurrence is not aligned along block boundaries. Our findings agree with and confirm prior understanding in the literature [18].

Average Chunk Size: Deduplication systems for backup data, such as in [26], use typical average chunk sizes of 4 or 8KB. The use of smaller chunk sizes can give higher space savings, arising from duplicate detection at finer granularities. On the flip side, this is associated with larger index sizes and increased chunk metadata. Moreover, when the chunk is compressed, usually by a

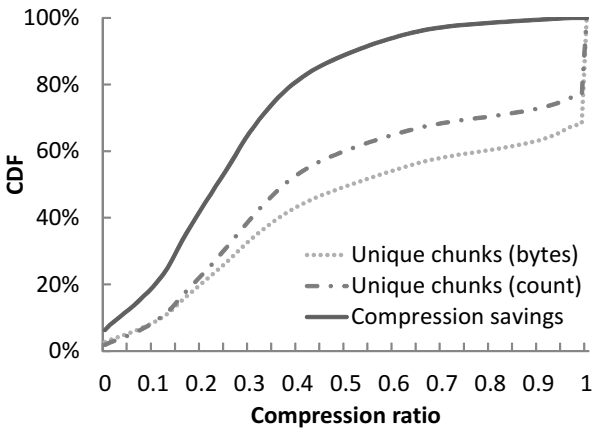


Figure 3: Chunk compressibility analysis for GFS-US dataset.

LZW-style [25] dictionary based lossless compressor, the smaller chunk size leads to reduced compression performance, as the adaptive dictionary in the lossless compressor has to be reset for each chunk and is not allow to grow to significant size. The adverse impact of these effects is more pronounced for a primary data deduplication system that is also serving live data.

We resolve the conflict by looking for reasonable tradeoffs between larger average chunk sizes and higher deduplication space savings. In Figure 2, we plot the deduplication space savings with and without chunk compression on the GFS-US dataset when the average chunk size is varied from 4KB to 64KB. We see that high deduplication savings achieved with 4KB chunk size are attainable with larger 64KB chunks with chunk compression, as the loss in deduplication opportunities arising from use of larger chunk sizes is canceled out by increased compressibility of the larger chunks.

Chunk compressibility: In Figure 3, we examine the distribution of chunks by compression ratio for the GFS-US dataset. We define the compression ratio c as (size of compressed chunk)/(size of original chunk). Therefore, a chunk with $c = 0.7$ saves 30% of its space when compressed. On the x-axis, we sort and bin unique chunks in the dataset by their compression ratio. On the y-axis, we plot the cumulative distribution of those unique chunks by both count and bytes. We also plot the cumulative distribution of the compression savings (bytes) contributed by those unique chunks across the same compression ratio bins.

We find significant skew in where the compression savings come from – 50% of the unique chunks are responsible for 86% of the compression savings and those chunks have a compression ratio of 0.5 or lower. On the other hand, we find that roughly 31% of the chunks (42% of the bytes) do not compress at all (i.e., fall at compression ratio = 1.0). This result is consistent across all datasets studied (including subsets of datasets created by grouping files of the same type) and it implies that it is feasible for a primary data deduplication system to be selective when compressing chunks. By checking compression ratios and storing compressed only those chunks whose compression ratios meet a certain threshold, the amount of decompression involved during data access can be reduced (assuming each chunk is equally likely to be accessed which is at least true for full file read IO requests). This allows for capturing most of the savings of chunk compression while eliminating the cost of decompression on the majority of chunks.

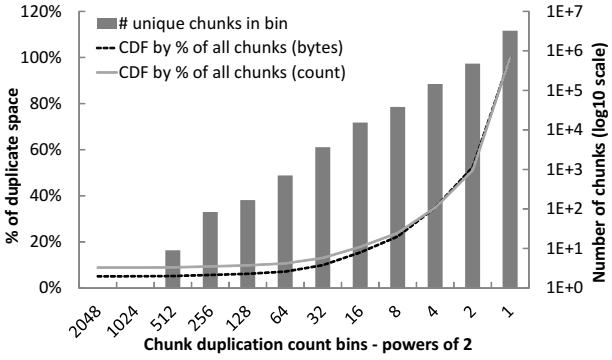


Figure 4: Chunk duplication analysis for GFS-Japan-1 dataset.

Chunk duplication analysis: In Figure 4, we examine the distribution of chunk duplication in primary data. On the x-axis, we bin the chunks by number of times they are duplicated in the dataset in power of 2 intervals. The rightmost bin contains unique chunks. Thereafter, moving towards the left, each bin contains chunks duplicated in the interval $[2^i, 2^{i+1})$ for $i \geq 1$. On the y-axis, the bars represent the total number of unique chunks in each bin and the CDFs show the distribution of unique chunks by both count and bytes across the bins. First, we see that about 50% of the chunks are unique, hence a primary data deduplication system should strive to reduce the overhead for serving unique data. Second, the majority of duplicate bytes reside in the middle portion of the distribution (between duplication bins of count 2 and 32), hence it is not sufficient to just deduplicate the top few bins of duplicated chunks. This points to the design decision of *deduplicating all chunks that appear more than once*. This trend was consistent across all datasets.

Data partitioning: We examine the impact of partitioning on each dataset by measuring the deduplication savings (without compression) within partitions and across partitions. We partition each dataset using two methods, namely (i) partitioning by file type (extension), and

Dataset	Dedup Space Savings		
	Global	Clustered by	
		File type	File path
GFS-US	36.7%	35.6%	24.3%
GFS-Japan-1	41.1%	38.9%	32.3%
GFS-Japan-2	39.1%	36.7%	24.8%
HF-Amsterdam	15.2%	14.7%	13.6%
HF-Dublin	16.8%	16.2%	14.6%
HF-Japan	19.6%	19.0%	12.9%

Table 3: Space savings for global dedup vs. dedup within partitioned file clusters, by file type or file path similarity. (See text on why some datasets were excluded.)

Dataset	Total Size	Per-Server Dedup Savings	Cross-Server Dedup Savings	Cross-Server Dedup Benefit
All Home-Folder Srvrs	2438GB	386GB	424GB	1.56%
All File-Share Srvrs	2897GB	1075GB	1136GB	2.11%
All Japan Srvrs	1436GB	502GB	527GB	1.74%
All US Srvrs	3844GB	1292GB	1354GB	1.61%

Table 4: Dedup savings benefit of cross-server deduplication, as fraction of original dataset size.

(ii) partitioning by directory hierarchy where partitions correspond to directory subtrees with total bytes at most 10% of the overall namespace. We excluded two datasets (Virtualization and Sharepoint) because all or most files were of one type or had a flat directory structure, so no meaningful partitioning could be done for them. The remaining datasets produced partitions whose size varied from one-third to less than one-tenth of the dataset.

As seen in Table 3, we find the loss in deduplication savings when partitioning by file type is negligible for all datasets. On the other hand, we find partitioning by directory hierarchy to be less effective in terms of dedup space savings.

Since the original datasets were naturally partitioned by server, we then inspect the effect of merging datasets to find out the impact of deduplication savings across servers. In Table 4, we combine datasets both by workload type and location. We find that the additional savings of cross-server deduplication to be no more than 1-2% above per-server savings.

This implies that it is feasible for a deduplication system to reduce resource consumption by performing partitioned deduplication while maintaining comparable space savings.

4 System Design

We first enumerate key requirements for a primary data deduplication system and discuss some design implications arising out of these and the dataset analysis in Section 3. We then provide an overview of our system. Specific aspects of the system, involving data chunking, chunk indexing, and data partitioning are discussed next in more detail.

4.1 Requirements and Design Implications

Data deduplication solutions have been widely used in backup and archive systems for years. Primary data deduplication systems, however, differ in some key workload constraints, which must be taken into account when designing such systems.

1. **Primary data:** As seen in Section 3, primary data has less duplication than backup data and more than 50% of the chunks could be unique.

2. **Primary workload:** The deduplication solution must be able to deduplicate data as a background workload since it cannot assume dedicated resources (CPU, memory, disk I/O). Furthermore, data access must have low latency - ideally, users and applications would access their data without noticing a performance impact.

3. **Broadly used platform:** The solution cannot assume a specific environment – deployment configurations may range from an entry-level server in a small business up to a multi-server cluster in an enterprise. In all cases, the server may have other softwares installed, including software solutions which change data format or location.

Based on these requirements and the dataset analysis in Section 3, we made some key design decisions for the system which we outline here.

Deduplication Granularity: Our earlier data analysis has shown that for primary datasets, whole file and sub-file fixed-size chunk deduplication were significantly inferior to sub-file variable size chunk deduplication. Furthermore, we learned that chunk compression yields significant additional savings on top of deduplication with greater compression savings on larger chunks, hence closing the deduplication savings gap with smaller chunk sizes. Thus, an average chunk size of about 80KB (and size range 32-128KB) with compression yields savings comparable to 4KB average chunk size with compression. Maximizing savings and minimizing metadata are both highly desirable for primary data

deduplication where tighter system resource constrains exist and there's less inherent duplication in the data. Our system uses variable size chunking with optional compression, and an average chunk size of about 80KB.

Inline vs. Post-processing Deduplication: An important design consideration is when to deduplicate the data. Inline deduplication processes the data synchronously on the write path, before the data is written to disk; hence, it introduces additional write latencies and reduces write throughput. On a primary data server, however, low write latency is usually a requirement as writes are common, with typical 1:3 write/read ratios [15].

Post-processing deduplication processes the data asynchronously, after it has been written to disk in its original format. This approach has the benefit of applying time-based policies to exploit known file access patterns. On file servers, most files are not accessed after 24 hours from arrival [15]. Therefore, a solution which deduplicates only older files may avoid additional latency for most accessed files on the server. Furthermore, post processing deduplication has the flexibility to choose when (e.g., idle time) to deduplicate data.

Our system is based on a post-processing approach, where the agility in deduplication and policies is a better fit for a primary data workload than inline approach.

Resource Usage Scaling with Data Size: One major design challenge a deduplication solution must face is scale and performance - how to scale to terabytes of data attached to a single machine, where (i) CPU memory and disk IOPS are scarce and used by the primary workload, (ii) the deduplication throughput must keep up with the data churn, (iii) dedicated hardware cannot be assumed and (iv) scale out to other machines is optional at best but cannot be assumed. Most, if not all, deduplication systems use a chunk hash index for identifying chunks that are already stored in the system based on their hash. The index size is proportional to the hash size and the number of unique chunks in the system.

Reducing the memory footprint for post-processing deduplication activity is a necessity in primary data servers so that enough memory is available for serving primary workloads. Common index designs in deduplication systems minimize the memory footprint of the chunk hash index by trading some disk seeks with using less memory. The index size is still relative to number of chunks, therefore, beyond a certain data scale, the index will just not fit within the memory threshold assigned to the deduplication system.

In our system, we address that level of scale by using two techniques that can work in concert or separately to reduce the memory footprint for the deduplication process. In the first technique, we use a low RAM footprint chunk hash index that uses about 6 bytes of RAM for every unique chunk stored in the system. This is discussed in more detail in Section 4.4.

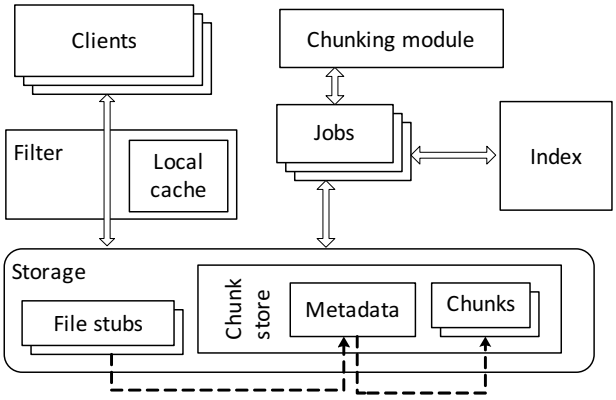


Figure 5: Deduplication engine architecture.

In the second technique, we partition the data into smaller buckets or partitions and deduplicate within each partition. Partitioning may be either permanent, in which case duplicate data may exist across partitions, or temporary, in which case data will be deduplicated across partitions in a subsequent phase called *reconciliation*. This implies that the deduplication process needs to scale only to the maximum partition size. This is discussed in more detail in Section 4.5.

4.2 System Architecture

In Sections 4.3, 4.4, and 4.5, we expand on the post-processing deduplication aspects of our system, including data chunking, chunk indexing, and data partitioning and reconciliation. To provide context for this discussion, we provide here an overview of our overall primary data deduplication system, as illustrated in Figure 5. The design aspects of our system related to primary data serving (beyond a brief overview in this section) have been left out to meet the paper length requirements. The detailed presentation and evaluation of those aspects is planned as a future paper.

Deduplication engine. The deduplication engine consists of a file system filter driver and a set of background jobs (post-processing deduplication, garbage collection, compaction, scrubbing) necessary to maintain continuous deduplication process and to maintain the integrity of the underlying store. The deduplication filter redirects the file system interactions (read, write, etc.) to transparently enable the same file semantics as for a traditional server. The background jobs are designed to run in parallel with the primary server IO workload. The engine actively monitors the primary work load (file serving) and the load of the background jobs. It automatically allocates resources and backs off background jobs to ensure

that these jobs do not interfere with the performance of the primary workload. Under the cover, the deduplication engine maintains the chunks and their metadata in large container files (chunk store) which enable fast sequential IO access on management operations.

Background jobs. The functionalities of background jobs in the deduplication engine are as follows. The post-processing deduplication job progressively scans the underlying volume and identifies candidate deduplication files, which are all files on the server that meet a certain deduplication policy criteria (such as file age). It scans the candidate file, chunks it into variable size chunks (Section 4.3), detects duplicate chunks using a chunk hash index (Section 4.4), and inserts unique chunks in the underlying chunk store, after an optional compression phase. Subsequently, the file itself is *transactionally* replaced with a virtual file stub, which contains a list of references to the associated chunks in the chunk store. Optionally, the stub may also contain extra references to sub-streams representing ranges of non-deduplicated data within the file. This is particularly useful when a deduplicated file is modified, as the modified portion can be represented as sub-streams of non-deduplicated data to support high speed and low latency primary writing operation. The candidate files are incrementally deduplicated – progress state is periodically saved to minimize expensive restarts on a large file in case of job interruptions such as job back-off (triggered by yielding resources to the primary workload) or cluster failover.

The garbage collection job periodically identifies orphaned chunks, i.e., chunks not referenced by any files. After garbage collection, the compaction job is periodically invoked to compact the chunk store container files, to reclaim space from unreferenced chunks, and to minimize internal fragmentation. Finally, the scrubbing job periodically scans the file stubs and the chunk store container files to identify and fix storage-level corruptions.

File reads. File reads are served differently depending on how much file data is deduplicated. For a regular, non-deduplicated file, reads are served from the underlying file system. For fully-deduplicated files, a read request is fulfilled in several steps. In the first step, a file-level read request is translated into a sequence of read requests for each of the underlying chunks through the file-level redirection table. This table essentially contains a list of chunk-IDs and their associated offsets within that particular file. In a second step, the chunk-ID is parsed to extract the index of its container and the virtual offset of the chunk body within the container. In some cases, if the container was previously compacted, the virtual chunk offset is not identical with the physical offset – in that case, a secondary address translation is done through another per-container redirection table. Finally, the read request data is re-assembled from the contents of the corresponding chunks. A similar sequence of operations applies to partially-deduplicated files, with one important difference – the system maintains an extra per-file bitmap to keep track of the non-deduplicated regions within the file, which are kept in the original file contents as a "sparse" stream.

File writes. File writes do not change the content of the chunk store. A file write simply overwrites a range of bytes in the associated sparse stream for that file. After this operation, some chunks may hold data that is old (obsolete) with respect to this file. No explicit deallocation of these chunks is done during the write, as these chunks will be garbage collected later (provided they are not referenced by any other file in the system). The deduplication filter has the metadata and logic to rebuild a file from ranged allocated in the sparse stream and ranges backed up by chunks in the chunk store.

4.3 Data Chunking

The chunking module splits a file into a sequence of chunks in a content dependent manner. A common practice is to use Rabin fingerprinting based sliding window hash [22] on the data stream to identify chunk boundaries, which are declared when the lower order bits of the Rabin hash match a certain pattern P. The length of the pattern P can be adjusted to vary the average chunk size. For example, we use a $|P| = L = 16$ bit pattern, giving an average chunk size of $S_{avg} = 64KB$.

Minimum and maximum chunk sizes. It is desirable for the chunking module in a primary dedup system to generate a chunk size distribution where both very small and very large chunks are undesirable. The very small sized chunks will lead to a larger number of chunks to be indexed, which increases the load of the indexing module. Moreover, for small chunks, the ratio between the chunk metadata size to chunk size is high, leading to performance degradation and smaller dedup savings. The very large sized chunks may exceed the allowed unit cache/memory size, which leads to implementation difficulties in other parts of the dedup systems.

A standard way to avoid very small and very large chunk sizes is to use S_{min} and S_{max} thresholds for minimum and maximum chunk sizes respectively. To enforce the former, a chunk boundary within S_{min} bytes of the last chunk boundary is simply suppressed. The latter is enforced by declaring a chunk boundary at S_{max} bytes when none has been found earlier by the hash matching rule.

One consequence of this size dependent boundary enforcement is the accumulation of peaks around S_{min} and S_{max} in the chunk size distribution and the possible reduction in dedup space savings due to declaration of chunk boundaries that are not content dependent. For example, with $S_{max} = 2 \times S_{avg}$, it can be shown that for ran-

dom data, 14% of chunks will not find a chunk boundary within S_{max}, which is a significant fraction. We design a *regression chunking algorithm* that aims to reduce the peaks in the chunk size distribution around S_{max}, while preserving or even improving the dedup space savings.

Reducing forced boundary declaration at maximum chunk size. The basic idea involves relaxing the strict pattern matching rule – we allow matches on suffixes of the bit pattern P so as to avoid forced boundary declaration at maximum chunk size when possible. Let k denote the maximum number of prefix bits in the pattern P whose matching can be relaxed. Then, we attempt to match the last $L-i$ bits of the pattern P with the lower order bits of the Rabin hash, with *decreasing priority* for $i = 0, 1, \ldots, k$. A boundary is declared at maximum chunk size *only* when no relaxed suffixes match after S_{max} bytes are scanned. For ties within the same suffix match, the later match position in the sequence is used.

In summary, this technique enables the maximum chunk size bound to be satisfied in a content dependent manner *more often* through gradual regression to larger match sets with smaller number of matching bits. With $k = 1$ level of regression, the probability of forcing a chunk boundary at S_{max} for random data reduces to 1.8%. At $k = 4$ levels of regression, the probability is extremely low at 10^{-14}. We use this value of k in our system.

Chunking throughput performance. To maintain fast chunking throughput performance, it is crucial not to break out of the core matching loop often, or increase the complexity of the core loop through additional comparisons. By using regression chunking with nested matching bits, we need to match against only the smallest $(L-k)$-bit suffix of the pattern in the core loop. Only when the smallest suffix match is satisfied do we break out of the core loop and further evaluate the largest suffix match. Regression chunking can be implemented by forward processing the data only once (without ever needing to track back); the only additional state needed is the last match position for each relaxed suffix match.

For the case $k = 2$, regression chunking has some similarities with the TTTD chunking algorithm [13] which uses a second smaller size pattern to declare chunk boundaries when a maximum chunk size is reached. Regression chunking uses additional match patterns to reduce the probability of forcing a chunk boundary at S_{max} to a very small number. Moreover, by making the match patterns progressive suffixes of a base pattern, it maintains fast chunking performance as explained above.

4.4 Chunk Indexing

In a typical deduplication system, the hash index is a design challenge for scale and resource consumption since the index size is relative to the data size. On a primary data deduplication system, conserving memory and disk IOPS resources is crucial. This paper discusses two methods for addressing the index scale. In this section, we present a hash index designed to have a low memory footprint and use reduced read/write disk IOPS so that it can scale to larger datasets and deliver high insert/lookup performance. It incorporates ideas from ChunkStash [10], but the use of flash memory for index storage is not mandatory. In the next section, we present a design for partitioning the index. The two methods can be used independently or in conjunction with each other. The index design is summarized in Figure 6.

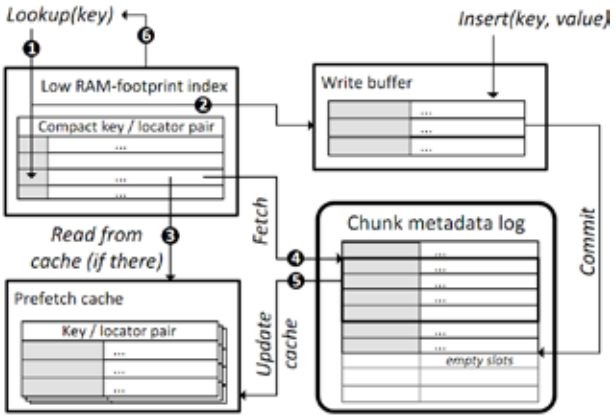


Figure 6: Chunk hash index.

Log-structured organization. The chunk metadata records, comprising of SHA-256 hash and location information, are organized in a log-structured manner on secondary storage. Newly inserted chunk records are held in a write buffer in RAM and *appended* to this log in a *batched* manner so that write IOPS are amortized across multiple index insert operations.

Low RAM footprint index. A specialized in-memory hash table is used to index chunk metadata records on secondary storage, with hash collisions resolved by a variant of cuckoo hashing [20]. The in-memory hash table stores 2-byte compact key signatures instead of full 32-byte SHA-256 chunk hashes so as to strike tradeoffs between RAM usage and false secondary storage reads. The compressed key signature also serves to eliminate disk accesses for lookups on non-existent chunks (with very low false positives). The index footprint is extremely low at about 6 bytes of RAM per chunk.

Prefetch cache. This technique aims to reduce disk IOPS during lookups and has been used for backup data dedup in [26]. When a chunk record R is read from the log, a total of $p = 1000$ sequential records are read in *one single IO*, starting from the location of record R, and inserted into a *prefetch cache* maintained in RAM. While sequential predictability of chunk hash lookups for primary data dedup is expected to be much less than that

for backup data dedup, we did not use smaller values of p since disk seek times dominate for prefetching. The prefetch cache is sized to contain $c = 100,000$ entries, which consumes (an acceptable) 5MB of RAM. It uses an LRU replacement policy.

Disk accesses for index lookups. An index lookup hits the disk *only when* the associated chunk is a duplicate *and* is not present in the prefetch cache (modulo a low false positive rate). The fraction of index lookups hitting disk during deduplication is in the 1% ballpark for all evaluated datasets.

4.5 Data Partitioning and Reconciliation

In Section 4.1, we motivated the need to reduce memory footprint for scale and presented partitioning as one of the techniques used. Partitioning scopes the deduplication task to a smaller namespace, where the namespace size is controlled by the deduplication engine. Therefore, the resource consumption (RAM, IOPS) can be very precise, possibly provided as an external threshold to the system. In this section, we discuss a design for deduplicating within partitions and then reconciling the partitions. The space savings are comparable to deduplicating the entire namespace without partitioning.

Data Partitioning Methods. In Section 3, we demonstrated that partitioning by server or file type is effective - most of the space savings are achieved by deduplicating within the partition, additional savings by deduplicating across partitions are minimal. Other methods may partition by namespace (file path), file time, or file similarity fingerprint. Partitions need to be constructed such that the maximum partition size can be indexed within a defined memory threshold. As the system scales, the number of partitions grow, but the memory consumption remains constant. A solution may consider a hybrid hierarchical partitioning technique, where permanent partitioning is applied at one level and then temporary partitioning is applied at the next level when indexing a permanent partition is too large to fit in memory. The temporary partitions may then be reconciled while the permanent partitions are not. In our design, we use permanent partitioning based on servers, then apply temporary partitioning based on file type or file time bound by a memory threshold. We then reconcile the temporary partitions with an efficient reconciliation algorithm, which is memory bound and utilizes large sequential reads to minimize IOPS and disk seeks.

Two-phase Deduplication. We divide the deduplication task into two phases:

1. *Deduplication within a partition:* The deduplication process is similar to deduplicating an entire dataset except that the hash index loads only the hashes belonging to chunks or files within the partition. We divide the hashes into partitions based on either file type or file age and then, based on memory threshold, we load hashes to the index such that the index size is within the threshold. In this deduplication phase, new chunks ingested into the system are deduplicated only if they repeat a chunk within the partition. However, if a new chunk repeats a chunk in another partition, the duplication will not be detected and the chunk will be stored in the system as a new unique chunk.

2. *Reconciliation of partitions:* Reconciliation is the process of deduplication across partitions to detect and remove duplicate chunks resulting from the previous phase. The design for the reconciliation process has the same goals of minimizing RAM footprint and disk seeks. Given two partitions, the reconciliation process will load the hashes of one partition (say, partition 1) into a hash index, and then scan the chunks belonging to the second partition (say, partition 2), using sequential I/O. We assume that the chunk store is designed to allow sequential scan of chunks and hashes. For each chunk of partition 2, the reconciliation process looks up the hash in the index loaded with all the hashes of partition 1 to detect duplication. One efficient way of persisting the detected duplicates is to utilize a merge log. The merge log is a simple log where each entry consists of the two chunk ids that are a duplicate of each other. The engine appends to the merge log with sequential writes and may batch the writing of merge log entries for reducing IOPS further.

   Once the chunk duplication is detected, reconciliation uses a chunk-merge process whose exact implementation depends on the chunk store. For example, in a chunk store that stores chunks within container files, chunk-merge process may read the merge log and then create new revision of the containers (using sequential read and write), omitting the duplicate chunks. At the end of the second phase, there are no duplicate chunk instances and the many partitions turn into a single reconciled partition.

Reconciliation Strategies. While reconciliation is efficient, it still imposes some I/O overhead on the system. The system may self-tune based on availability of resources and idle time whether to reconcile all partitions or selectively. In the case that reconciliation across all partitions is the selected strategy, the following method is used. The basic idea is to consider some number of unreconciled partitions at a time and grow the set of reconciled partitions by comparing the current group of unreconciled partitions to each of those in the already reconciled set. This is done by indexing in memory the

current group of unreconciled partitions and then scanning sequentially the entire reconciled set of partitions and building merge logs for all. This process repeats itself until all unreconciled partitions join the reconciled set. The number of unreconciled partitions considered in each iteration depends on the amount of available memory – this allows a tradeoff between the speed of the reconciliation process and the amount of memory used.

Our reconciliation strategy above uses the *hash join* method. An alternative method is *sort-merge* join, which is not only more CPU intensive (since sorting is more expensive than building a hash table) but is also disk IO expensive (as it requires reading and writing buckets of sorted partitions to disk).

Selective reconciliation is a strategy in which only some subsets of the partitions are reconciled. Selecting which partitions to reconcile may depend on several criteria, such as file types, signature computed from the data within the partition, or some other criteria. Selective reconciliation is a tradeoff between reducing IOPS and achieving maximal deduplication savings.

Another useful strategy is delayed reconciliation – rather than reconciling immediately after deduplication phase, the deduplication engine may defer it to server's idle time. With this strategy, the tradeoff is with how long it takes to achieve maximal savings. As seen in Section 3, deduplication across (appropriately chosen) partitions yields incremental additional savings, therefore both selective or delayed reconciliation may be the right tradeoff for many systems.

5 Performance Evaluation

In Section 3, we analyzed our datasets around multiple aspects for dedup space savings and used those findings to design our primary data deduplication system. In this section, we evaluate some other aspects of our primary data deduplication system that are related to post-processing deduplication.

Post-processing deduplication throughput. Using the GFS-US dataset, we examined post-processing deduplication throughput, calculated as the amount of original data processed per second. An entry-level HP ProLiant SE326M1 system with one quad-core Intel Xeon 2.27 GHz L5520 and 12 GB of RAM was used, with a 3-way RAID-0 dynamic volume on top of three 1TB SATA 7200 RPM drives.

To perform an apples-to-apples comparison, we ran a post-processing deduplication session multiple times with different indexing options in a controlled environment. Each deduplication session uses a *single thread*. Moreover, we ensured that there were no CPU-intensive tasks running in parallel for increased measurement accuracy. The baseline case uses a *regular index* where the full hash (SHA-256, 32 bytes) and location information (16 bytes) for each unique chunk is stored in RAM. The *optimized index* uses the RAM space efficient design described in Section 4.4. The number of data partitions for this workload was chosen as three by the system according to a implementation heuristic; however, partitioning could be done with as many partitions as needed. The resource usage and throughput numbers are summarized in Table 5. The following important observations can be drawn from Table 5:

1. **RAM frugality:** Our optimized index reduces the RAM usage by about 8x. Data partitioning can reduce RAM usage by another 3x (using 3 partitions as chosen for this workload). *The overall RAM usage reduction with optimized index and data partitioning is as much as 24x.*

2. **Low CPU utilization:** The median *single core* utilization (measured over a single run) is in the 30-40% range for all four cases. Compared to the baseline case, it is slightly higher with optimized index and/or data partitioning because of indexing work and reconciliation respectively. Note that modern file servers use multi-core CPUs (typically, quad core or higher), hence this leaves enough room for serving primary workload.

3. **Low disk usage:** (not shown in Table 5) The median disk queue depth is zero in all cases. At the 75-th percentile, the queue depth increases by 2 or 3 as we move from baseline to optimized index and/or data partitioning. In the optimized index case, the increase is due to extra time spent in disk-based index lookups. With data partitioning, the increase is mainly due to reconciliation (which uses mostly sequential IOs).

4. **Sustained deduplication throughput:** Even as RAM usage goes down significantly with optimized index and data partitioning, *the overall throughput performance remains mostly sustained in the range of 26-30 MB/sec*, with only about a 10% decrease for the lowest RAM usage case. This throughput is sufficient to keep up with data ingestion rate on typical file servers, which is small when compared to total stored data. In a four month dynamic trace study on two file servers hosting 3TB and 19TB of data, Leung et al. [15] reported that 177.7GB and 364.4GB of data was written respectively. This computes to an average ingestion rate of 0.03 MB/sec and 0.04 MB/sec respectively, which is *three orders of magnitude lower* than the obtained single deduplication session throughput.

	Regular Index (Baseline)	Optimized index	Regular index w/ partitions	Optimized index w/ partitions
Throughput (MB/s)	30.6	28.2	27.6	26.5
Partitioning factor	1	1	3	3
Index entry size (bytes)	48	6	48	6
Index memory usage	931MB	116MB	310MB	39MB
Single core utilization	31.2%	35.2%	36.8%	40.8%

Table 5: Deduplication processing metrics for regular and optimized indexes, without and with data partitioning, for GFS-US dataset. Disk queue depth (median) is zero in all cases and is not shown.

The above two observations confirm that we meet our design goal of *sustained post-processing deduplication performance at low overhead so that server memory, CPU, and disk resources remains available for serving primary workload*. We also verified that *deduplication processing is parallelizable across datasets and CPU cores/disks* – when datasets have disk diversity and the CPU has at least as many cores as dataset deduplication sessions, the aggregate deduplication throughput scales as expected, assuming sufficient RAM is available.

It is also interesting to note that the extra lookup time spent in the optimized index configuration (for lookups going to disk) is comparable with the time spent in reconciliation in the regular index with partitioning case. To explore this further, we have compared the contribution of various sub-components in the deduplication session in Table 6. As can be seen, in the case of optimized index without partitioning, the main impact on throughput reduction comes from the index lookup time, when lookups miss in the prefetch cache and hit disk. In the case of data partitioning with regular index, the index lookup time is greatly reduced (at the cost of additional memory) but the main impact is deferred to the partition reconciliation phase. We also observe that the CPU time taken by the data chunking algorithm remains low compared with the rest of the post-processing phase.

Chunk size distribution. We compare the chunk size distribution of the regression chunking algorithm described in Section 4.3 with that of basic chunking on the GFS-US dataset. In Figure 7, we see that regression chunking achieves a more uniform chunk size distribution – it flattens the peak in the distribution around maximum chunk size (128KB) (by relaxing the chunk boundary declaration rule) and spreads out the distribution more uniformly between minimum and maximum chunk sizes. Regression chunking obtains an average chunk size of 80KB on this dataset, which is a 5% decrease over that obtained by basic chunking. This can also be attributed to the effect discussed above.

Deduplication Activity	Optimized index	Regular index w/ partitioning
Index lookup	10.7%	0.4%
Reconciliation	n/a	7.0%
Compression	15.1%	15.3%
SHA hashing	14.3%	14.6%
Chunking	9.7%	9.7%
Storing unique data	11.3%	11.5%
Reading existing data	12.6%	12.8%

Table 6: Percentage time contribution to the overall post-processing session for each component of deduplication activity for GFS-US dataset.

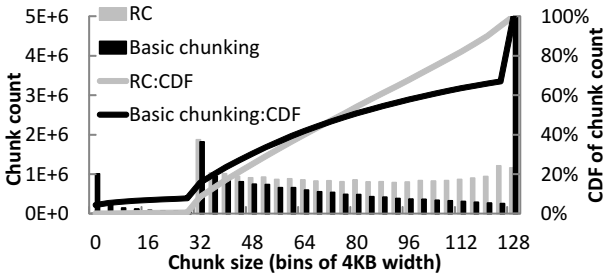


Figure 7: Distribution of chunk size for GFS-US dataset.

In Table 7, we plot the improvement in dedup space savings obtained by regression chunking over basic chunking on this dataset. Although the overall improvement is about 3%, we see significant improvements for some file types contained in that dataset – for example, the dedup savings increases by about 27% for pdf file types. Thus, depending on the mix of file types in the dataset, regression chunking can provide marginal to significant additional dedup space savings. This effect can be attributed to regression chunking declaring more

Dataset	Dedup Space Savings		
	Basic Chunking	Regression Chunking (RC)	RC Benefit
Audio-Video	2.98%	2.98%	0%
PDF	9.96%	12.70%	27.5%
Office-2007	35.82%	36.65%	2.3%
VHD	48.64%	51.39%	5.65%
GFS-US	36.14%	37.2%	2.9%

Table 7: Improvement in dedup space savings with regression chunking over basic chunking for GFS-US dataset. Effect of chunk compression is *not* included.

chunk boundaries in a content dependent manner instead of forcing them at the maximum chunk size.

6 Conclusion

We presented a large scale study of primary data deduplication and used the findings to inform the design of a new primary data deduplication system implemented in the Windows Server 2012 operating system. We found duplication within primary datasets to be far from homogenous, existing in about half of the chunk space and naturally partitioned within that subspace. We found chunk compressibility equally skewed with the majority of compression savings coming from a minority of the chunks.

We demonstrated how deduplication of primary file-based server data can be significantly optimized for both high deduplication savings and minimal resource consumption through the use of a new chunking algorithm, chunk compression, partitioning, and a low RAM footprint chunk index.

We presented the architecture of a primary data deduplication system designed to exploit our findings to achieve high deduplication savings at low computational overhead. In this paper, we focused on aspects of the system which address scaling deduplication processing resource usage with data size such that memory, CPU, and disk resources remain available to fulfill the primary workload of serving IO. Other aspects of our system related to primary data serving (beyond a brief overview), reliability, and resiliency are left for future work.

Acknowledgements. We thank our shepherd Emmett Witchel, the anonymous reviewers, and Mathew Dickson, Cristian Teodorescu, Molly Brown, Christian Konig and Mark Manasse for their feedback. We acknowledge Young Kwon from Microsoft IT for enabling real-life production data to be analyzed. We also thank members of the Windows Server Deduplication project team: Jim Benton, Abhishek Gupta, Ian Cheung, Kashif Hasan, Daniel Hefenbrock, Cosmin Rusu, Iuliu Rus, Huisheng Liu, Nilesh Shah, Giridhar Kasirala Ramachandraiah, Fenghua Yuan, Amit Karandikar, Sriprasad Bhat Kasargod, Sundar Srinivasan, Devashish Delal, Subhayan Sen, Girish Kalra, Harvijay Singh Saini, Sai Prakash Reddy Sandadi, Ram Garlapati, Navtez Singh, Scott Johnson, Suresh Tharamal, Faraz Qadri, and Kirk Olynyk. The support of Rutwick Bhatt, Gene Chellis, Jim Pinkerton, and Thomas Pfenning is much appreciated.

References

[1] Linux SDFS. `www.opendedup.org`.

[2] NetApp Deduplication and Compression. `www.netapp.com/us/products/platform-os/dedupe.html`.

[3] Ocarina Networks. `www.ocarinanetworks.com`.

[4] Permabit Data Optimization. `www.permabit.com`.

[5] Windows Storage Server. `technet.microsoft.com/en-us/library/gg232683(WS.10).aspx`.

[6] ZFS Deduplication. `blogs.oracle.com/bonwick/entry/zfs_dedup`.

[7] BOLOSKY, W. J., CORBIN, S., GOEBEL, D., AND DOUCEUR, J. R. Single instance storage in windows 2000. In *4th USENIX Windows Systems Symposium* (2000).

[8] BRODER, A., AND MITZENMACHER, M. Network Applications of Bloom Filters: A Survey. In *Internet Mathematics* (2002).

[9] CLEMENTS, A., AHMAD, I., VILAYANNUR, M., AND LI, J. Decentralized Deduplication in SAN Cluster File Systems. In *USENIX ATC* (2009).

[10] DEBNATH, B., SENGUPTA, S., AND LI, J. ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory. In *USENIX ATC* (2010).

[11] DUBNICKI, C., GRYZ, L., HELDT, L., KACZMARCZYK, M., KILIAN, W., STRZELCZAK, P., SZCZEPKOWSKI, J., UNGUREANU, C., , AND WELNICKI, M. HYDRAstor: a Scalable Secondary Storage. In *FAST* (2009).

[12] EMC CORPORATION. EMC Centera: Content Addresses Storage System, Data Sheet, April 2002.

[13] ESHGHI, K. A framework for analyzing and improving content-based chunking algorithms. *HP Labs Technical Report HPL-2005-30 (R.1)* (2005).

[14] KRUUS, E., UNGUREANU, C., AND DUBNICKI, C. Bimodal Content Defined Chunking for Backup Streams. In *FAST* (2010).

[15] LEUNG, A. W., PASUPATHY, S., GOODSON, G., AND MILLER, E. L. Measurement and analysis of large-scale network file system workloads. In *USENIX ATC* (2008).

[16] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZISE, G., AND CAMBLE, P. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *FAST* (2009).

[17] MEYER, D. T., AND BOLOSKY, W. J. A study of practical deduplication. In *FAST* (2011).

[18] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. In *SOSP* (2001).

[19] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, FIPS 180-1. Secure Hash Standard. U.S. Department of Commerce, 1995.

[20] PAGH, R., AND RODLER, F. F. Cuckoo hashing. *Journal of Algorithms 51*, 2 (May 2004), 122–144.

[21] QUINLAN, S., AND DORWARD, S. Venti: A New Approach to Archival Data Storage. In *FAST* (2002).

[22] RABIN, M. O. Fingerprinting by Random Polynomials. *Harvard University Technical Report TR-15-81* (1981).

[23] SCHLEIMER, S., WILKERSON, D. S., AND AIKEN, A. Winnowing: Local algorithms for document fingerprinting. In *ACM SIGMOD* (2003).

[24] SRINIVASAN, K., BISSON, T., GOODSON, G., AND VORUGANTI, K. iDedup: Latency-aware, Inline Data Deduplication for Primary Storage. In *USENIX FAST* (2012).

[25] WELCH, T. A technique for high-performance data compression. *IEEE Computer* (June 1984).

[26] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *FAST* (2008).

Design and Implementation of an Embedded Python Run-Time System

Thomas W. Barr Rebecca Smith Scott Rixner
Rice University
{*twb, rjs, rixner*}*@rice.edu*

Abstract

This paper presents the design and implementation of a complete embedded Python run-time system for the ARM Cortex-M3 microcontroller. The Owl embedded Python run-time system introduces several key innovations, including a toolchain that is capable of producing relocatable memory images that can be utilized directly by the run-time system and a novel foreign function interface that enables the efficient integration of native C code with Python.

The Owl system demonstrates that it is practical to run high-level languages on embedded microcontrollers. Instrumentation within the system has led to an overall system design that enables Python code to be executed with low memory and speed overheads. Furthermore, this paper presents an evaluation of an autonomous RC car that uses a controller written entirely in Python. This demonstrates the ease with which complex embedded software systems can be built using the Owl infrastructure.

1 Introduction

For every microprocessor in a traditional computer system, there are dozens of microcontrollers in cars, appliances, and consumer electronics. These systems have significant software requirements, as users demand elaborate user interfaces, networking capabilities, and responsive controls. However, the programming environments and run-time systems for microcontrollers are extremely primitive compared to conventional computer systems.

Modern microcontrollers are almost always programmed in C, either by hand or generated automatically from models. This code, which runs at a very low level with no reliance on operating systems, is extremely difficult to debug, analyze, and maintain. At best, a simple real-time operating system (RTOS) is used to facilitate thread scheduling, synchronization, and communication [1, 3, 8, 11]. Typically, such RTOS's provide primitive, low-level mechanisms that require significant expertise to use and do very little to simplify programming. As the capabilities of embedded systems increase, this situation is becoming untenable. Programming must be simplified to meet the demand for increasingly complex microcontroller applications.

This paper presents one mechanism for doing so: an efficient embedded Python run-time system named *Owl*. The Owl system is a complete Python development toolchain and run-time system for microcontrollers that do not have enough resources to run a real operating system, but are still capable of running sophisticated software systems. These microcontrollers typically operate at 50–100 MHz, have 64–128 KB of SRAM, and have up to 512 KB of on-chip flash. One example of such a microcontroller is the ARM Cortex-M3. ARM predicts that in 2015, the market for these Cortex-M class microcontrollers will be around 18 billion units [7]. In contrast, Gartner, Inc. predicts that 404 million x86 processors will ship in 2012 [9].

Owl is a complete system designed for ARM Cortex-M microcontrollers that includes an interactive development environment, a set of profilers, and an interpreter. It is derived from portions of several open-source projects, including CPython and Baobab. Most notably, the core run-time system for Owl is a modified version of Dean Hall's Python-on-a-Chip (p14p).[1]

We have developed a comprehensive set of profilers and analysis tools for the Owl system. Using the data from these static and dynamic profiling tools, as well as experience from having a large user base at Rice, we significantly expanded, re-architected, and improved the robustness of the original p14p system. The Owl toolchain we developed includes simple tools to program the microcontroller on Windows, OS X, and Linux. We have also added two native function interfaces, stack protection, autoboxing, a code cache, and many other improvements to the run-time system. Furthermore, the toolchain and run-time system have been re-architected to eliminate the need for dynamic loading.

The Owl system demonstrates that it is possible to develop complex embedded systems software using a high-

[1] `http://code.google.com/p/python-on-a-chip/`

level programming language. Many software applications have been developed within the Owl system, including a GPS tracker, a web server, a read/write FAT32 file system, and an artificial horizon display. Furthermore, Owl is capable of running a soft real-time system, an autonomous RC car. These applications were written entirely in Python by programmers with no prior embedded systems experience, showing that programming microcontrollers with a managed run-time system is not only possible but extremely productive. Additionally, Owl is used as the software platform for Rice University's r-one educational robot [15]. A class of twenty-five first-semester students programmed their robots in Python without the interpreter ever crashing.

The cornerstone of this productivity is the interactive development process. A user can connect to the microcontroller and type statements to be executed immediately. This allows easy experimentation with peripherals and other functionality, making incremental program development for microcontrollers almost trivial. In a traditional development environment, the programmer has to go through a tedious compile/link/flash/run cycle repeatedly as code is written and debugged. Alternatively, in the Owl system a user can try one thing at the interactive prompt and then immediately try something else after simply hitting "return". The cost of experimentation is almost nothing.

Microcontrollers are incredibly hard to program. They have massive peripheral sets that are typically exposed directly to the programmer. As embedded systems continue to proliferate and become more complex, better programming environments are needed. The Owl system demonstrates that a managed run-time system for a high-level language is not only practical to implement for modern microcontrollers, but also makes programming complex embedded applications dramatically easier.

2 Related Work and Background

Early commercial attempts to build embedded run-time systems, such as the BASIC Stamp [12], required multiple chips and have not been used much beyond educational applications. Academic projects have largely focused on extremely small 8-bit devices [10, 13]. These systems are built to run programs that are only dozens of lines long and are simply not designed for more modern and capable 32-bit microcontrollers.

The Java Card system ran an embedded JVM subset to allow smartcards to perform some limited computation [6]. While there were some small proof-of-concept applications developed for it such as a basic web-server [16], the limited computational and I/O capabilities of smartcards rendered building large applications in Java Card impractical [17].

While Android uses an interpreter, Dalvik, that runs on embedded systems, it has very different design goals than these projects [5]. Dalvik relies on the underlying Linux kernel to provide I/O, memory allocation, process isolation and a file system. It is designed for systems with at least 64 MB of memory, three orders of magnitude more than is available on ARM Cortex-M microcontrollers.

Arduino[2] is a simple microcontroller platform that has been widely adopted by hobbyists and universities. It has shown that there is great interest in making programming microcontrollers easier. However, Arduino focuses on raising the abstraction level of I/O by providing high-level libraries, while Owl raises the abstraction level of computation with a managed run-time system.

Recently, two open-source run-time systems for high-level languages on microcontrollers have been developed: python-on-a-chip (p14p) and eLua. p14p is a Python run-time system that has been ported to several microcontrollers, including AVR, PIC and ARM. The p14p system is a portable Python VM that can run with very few resources and supports a significant fraction of the Python language. Similarly, eLua is a Lua run-time system that runs on ARM microcontrollers.[3] The overall objectives and method of operation of eLua are very similar to p14p, so they will not be repeated here.

The fundamental innovation of p14p is the read-eval-print-loop that utilizes the host Python compiler to translate source code into Python bytecodes at run-time. A p14p memory image is built from the compiled code object and then sent to the microcontroller. On the microcontroller, an image loader reads the image, creates the necessary Python objects, and then executes the bytecodes. In this manner, an interactive Python prompt operating on the host computer can be used to interact with the embedded run-time system over USB or other serial connection. This leads to an extremely powerful system in which microcontrollers can be programmed interactively without the typical compile/link/flash/run cycle. This process has been re-architected and improved in Owl, as described in Section 3.2.

The interactive Python prompt also gives unprecedented visibility into what is happening on the embedded system. Typically, a user is presented with a primitive command system that only enables limited interaction and debugging on the microcontroller. Debuggers, such as gdb, are needed for additional capability. In contrast, an interactive prompt allows users to run arbitrary code, print out arbitrary objects, and very easily understand the state of the system. This leads to much more rapid software development and debugging.

In p14p, native C functions can be wrapped in a Python function. This allows arbitrary C functions to be

[2] `http://arduino.cc/`

[3] `http://www.eluaproject.net/`

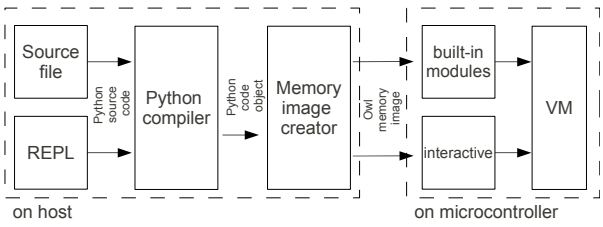


Figure 1: The Owl toolchain.

called from Python, enabling access to the microcontroller's peripherals. However, the C functions are specialized to p14p, use cryptic macros to access parameters and return values, and must be rewritten for every platform to which p14p is ported. The maintainers of p14p leave it to the user to figure out how to best provide access to I/O devices for their platform and application.

The Owl system is based upon a snapshot of p14p from April 2010. Using experience gained from having a large user base at Rice, we then significantly expanded, re-architected, and improved the robustness of p14p.

3 The Owl Toolchain

This section describes the Owl toolchain, as shown in Figure 1. The toolchain transforms code on the host into a form that is directly runnable on the microcontroller. Code starts on the host and is entered either into a file or an interactive prompt. It is compiled into a code object by the standard Python compiler, which is then transformed into a memory image. This image is copied into the microcontroller's flash memory. The images in flash can then be executed by the virtual machine, which will be described in Section 4.

3.1 Code sources

The Owl toolchain supports all of the functionality of p14p. Furthermore, it provides an interactive prompt over USB and drivers for Windows, OS X, and Linux. It also provides several additional capabilities that do not exist in p14p. Most notably, Owl includes a bootloader so that once the virtual machine has been programmed in flash, no C compiler or programming tools are needed by the user to program Python code onto the microcontroller. Directly from the Python interpreter on the host computer, the user can:

1. Type Python code at an interactive prompt connected to the virtual machine on the microcontroller. This code gets dynamically compiled on the host, stored in SRAM on the microcontroller, and is then executed immediately.
2. Load code from a Python source file on the host via the interactive prompt. The file gets dynamically compiled on the host, stored in SRAM on the microcontroller, and is then executed immediately.
3. Store code from one or more Python source files in flash on the microcontroller after being compiled on the host. This code can then be executed at any future time.

While p14p has the first capability, the latter two are unique to Owl. Therefore, one can program Owl systems using only Python without writing any C, needing a C compiler, or needing any specialized knowledge or hardware to program flash. Furthermore, code executed from all three sources can interact. At the command prompt, for instance, one can import a module that was previously stored in flash.

3.2 Memory images

Loaders and dynamic linkers are integral parts of traditional computer systems. They enable the compiler to generate code that is relocatable and can be combined with other libraries when the program is first run, as well as throughout its execution.

Dynamic loading and linking are also an integral part of run-time systems for most interpreted languages. For example, the desktop Python implementation (CPython) uses the marshal format to store compiled source files: each object used by the source is loaded from the file, wrapped in an object header, and placed on the Python heap. Java `.class` files are loaded similarly [14]. The p14p system also uses a similar architecture.

These design decisions are predicated on the assumption that programs cannot be directly executed off the disk and that memory is effectively infinite. On an embedded system, the situation is different. First, flash is fast enough (often with 1–2 cycle access times), that programs can be stored in, and executed directly from, flash. Second, memory (SRAM) is scarce. Therefore, it makes sense to do everything possible to keep programs in flash, copying as little as possible into SRAM.

To accomplish this, the Owl system architecture eliminates the need for dynamic code loading. Instead, the Owl toolchain compiles Python source code into a relocatable memory image, which contains all of the objects needed to run the user program. The run-time system then executes programs in these memory images directly from flash without copying anything to SRAM.

One of the key challenges in eliminating dynamic loading is handling compound objects which contain other objects. Compound objects created at run-time simply keep references to the objects they contain, which are located elsewhere in the heap. However, the compound objects within memory images cannot be handled in this way. They must be relocatable and therefore cannot contain references. In a traditional system with

a dynamic loader, such as p14p or Java, the compiler toolchain would generate special relocatable compound objects that are stored in a memory image. At run-time, the dynamic loader would first copy the relocatable compound object's constituent sub-objects from the memory image to the heap. Then, the loader would generate the compound object itself on the heap, populating it with references to the sub-objects.

To avoid these copies, Owl introduces *packed* tuples and code objects, which store objects internally rather than using references. Each internal object is a complete Python object, with all associated header and type information. The packed types therefore enable the internal objects to be referenced directly, completely eliminating the need to copy them into SRAM. The compiler toolchain never places compound objects that are not packed into a memory image, guaranteeing that a memory image is completely usable without any modification. The run-time system recognizes these packed objects and handles them appropriately. These novel compound objects are therefore both relocatable and directly usable without the need for dynamic loading. They also can be stored anywhere, including flash, SRAM, or an SD card, and can even be sent across a network.

4 The Owl Run-time System

The Owl run-time system executes the memory images prepared by the toolchain. It interprets bytecodes, manages memory and calls external functions through both wrapped functions and the embedded foreign function interface.

4.1 Python Interpreter

The main component of the run-time system is the interpreter, which executes Python bytecodes from the memory image. These bytecodes operate with one or more Python objects on a stack. For example, they may add values (BINARY_ADD), load a variable name (LOAD_NAME), or switch execution to a new code object (CALL_FUNCTION). The Owl interpreter is derived directly from p14p, uses bytecodes identical to CPython, and matches the overall structure of the CPython interpreter. The heap is managed by a mark-and-sweep garbage collector which automatically runs during idle periods and under memory pressure.

One of the key advantages of using an interpreter is that the virtual machine is the only code that can directly access memory. If the virtual machine and all native functions are stable, it is impossible for a user to crash the system. Additionally, error detection code can optionally be included throughout the system to ensure that bugs inside the interpreter are detected at run-time and reported as exceptions. Normally, such events would be silent, difficult to trace, errors. Partly because of these features, the Owl system itself does not crash, even though it has been heavily used.

The Owl interpreter also includes several additions to the original p14p interpreter. First, Owl includes stack protection, via optional run-time checks to ensure that stack frames are not overflowed and that uninitialized portions of the stack are not dereferenced. Second, Owl includes transparent conversion from basic types to objects through autoboxing. Basic types are automatically converted to an object when their attributes and methods are accessed. This means that basic types still have small memory overhead, since they don't generally need attribute dictionaries, but can be used like an object, as in traditional Python. Finally, Owl caches modules so that only one instance is ever present in memory. This saves considerable memory when multiple user modules include a common set of library modules.

4.2 Native C functions

While the use of Python on embedded systems provides enormous benefits in terms of productivity and reliability, it is necessary to write portions of many programs in C in order to to provide access to existing C libraries and to allow critical sections of code to run quickly. This is especially critical on a microcontroller where programs need to access memory-mapped peripherals via vendor-provided I/O libraries. For example, Texas Instruments provides a C interface to the entire peripheral set on their Cortex-M class microcontrollers, called StellarisWare, that simplifies the use of on-chip peripherals.

This section presents and compares two different techniques in Owl for allowing user code to call C functions: wrapped and foreign functions. While their implementations differ significantly, both systems make a native C library appear exactly like any other Python library.

While interpreters on the desktop have allowed programs to access external C libraries for some time, they have typically relied on features such as dynamic linking and run-time readable symbol tables that are too large for a microcontroller. In contrast, this section shows that a light-weight foreign function interface can be implemented in very little space without these features, and serve as an efficient bridge between Python and C code.

Providing the ability to execute native C functions introduces a way for the user to crash the system. However, all C functions must be compiled directly into the run-time system. Therefore, when discussing robustness and stability, they must be considered part of the run-time system itself. For peripheral and other library routines, such as StellarisWare, that are reused among applications, these functions are likely to be heavily tested and

```
/* Variable declarations */
PmReturn_t retval = PM_RET_OK;
pPmObj_t p0;
uint32_t peripheral;

/* If wrong number of arguments, raise TypeError */
if (NATIVE_GET_NUM_ARGS() != 1) {
  PM_RAISE(retval, PM_RET_EX_TYPE);
  return retval;
}

/* Get Python argument */
p0 = NATIVE_GET_LOCAL(0);

/* If wrong argument type, raise TypeError */
if (OBJ_GET_TYPE(p0) != OBJ_TYPE_INT) {
  PM_RAISE(retval, PM_RET_EX_TYPE);
  return retval;
}

/* Convert Python argument to C argument */
peripheral = ((pPmInt_t)p0)->val;

/* Actual call to native function */
SysCtlPeripheralEnable(peripheral);

/* Return Python object */
NATIVE_SET_TOS(PM_NONE);
return retval;
```

Figure 2: Body of autowrapped native function.

as stable as the rest of the run-time system. For application code that is ported to C for performance, special care must be taken to preserve the stability of the system.

4.2.1 Wrapped functions

A Python program calls a function by loading a callable Python object and executing the CALL_FUNCTION bytecode. The callable object can be a *Python* code object or a *native* code object which serves as an interface to a native C function. In p14p, the native functions themselves are responsible for pulling arguments off of the Python stack, checking argument types, executing the actual code of the function, and generating a Python object as a return value. Argument and return values are read/written via a set of C macros that provide access to the Python stack. With this interface, functions can be written in C and then accessed or called like any other Python object. In fact, p14p provides the ability to embed such C code in the document string of a Python function.

Figure 2 shows the C code required to wrap a call to the native function with the prototype:

```
void SysCtlPeripheralEnable(uint32_t peripheral);
```

In this function, one Python integer is first type checked and then converted into the variable `peripheral`. This variable is used as the argument for the call to `SysCtlPeripheralEnable`. Since this function does not return anything, the Python object `None` is pushed back on to the Python stack and the function returns.

Note that `SysCtlPeripheralEnable` could have been inlined, but was instead called indirectly for clarity. In this case, the underlying function is a StellarisWare function that manipulates hardware registers to enable an on-chip peripheral. This cannot be written in Python and is a prime example of the value of native functions.

Given that the argument and return value marshalling is tedious and mechanical, it is a prime target for automation. The Owl toolchain includes an *autowrapper*: an automated tool that generates a wrapper function for each function in a library. The wrapper is a small stub function that converts arguments from Python objects into C variables, calls the function, and, if necessary, converts the return value into a Python object and places it on the Python stack. In fact, the code shown in Figure 2 was generated by the autowrapper. Autowrapping functions is similar to the technique used by SWIG, which is commonly used to provide access to C code from high-level languages [4].

While this approach is conceptually simple, these conversions and type checks must be repeated for each function that is wrapped. This results in a massive amount of object code that is essentially repeated in the final binary.

4.2.2 Embedded foreign function interface

Given that the wrapper code can be generated mechanically, it is also a prime candidate for elimination. Foreign function interfaces, such as `libffi`[4], have been developed for precisely this reason: to enable access to native functions from an interpreted language. Typically, a C compiler generates the code necessary to call a function. When one function calls another, it includes code that places arguments and a return address into registers and/or the stack according to that platform's calling convention. Then, the address of the called function is loaded into the program counter. `libffi` does this process dynamically at run-time. A user can call `libffi` with a list of arguments and a pointer to a function; it then loads this data into registers and the stack and calls the given function. Java, Python, and PLT Scheme all use `libffi` to allow programmers to call external functions.

eFFI, a light-weight foreign function interface developed for the Owl system, builds upon these concepts in a fashion suitable for embedded run-time systems. It is a rewrite of `libffi` for the Cortex-M3 with some critical modifications. Specifically, embedded systems do not typically include the ability to dynamically link native code, so all native libraries must be statically linked. eFFI links target libraries when the run-time system is compiled, therefore requiring much less support from either the user or the host system.

First, the header file of the library that is accessed by user programs is read into a variant of the autowrapping tool discussed in Section 4.2.1. This tool reads the names

[4] `http://sourceware.org/libffi/`

and signatures of each function to be exposed. For each function, a Python callable object is generated containing argument types, return type, and a reference to the function itself. Since this object is generated automatically at compile time, the programmer never needs to specify the number or types of arguments, eliminating one possible source of error when using foreign functions.

The signatures and addresses of all the foreign functions to be exposed to the virtual machine are stored in a compact data structure. These are each made available to the user as *foreign function objects*, stored in flash.

When a foreign function object is called at run-time, each argument is converted into a C variable and placed onto the C stack or loaded into registers. The address of the function is then written into the program counter, jumping into the function. When the function returns, the result is copied off the stack or out of registers, converted into the proper Python type (as specified in the foreign function object) and pushed back onto the Python stack.

The key to the lightweight implementation of eFFI is that unlike the FFI implementations in desktop interpreters, foreign functions are not referenced by name in eFFI. Before the run-time system is compiled, arrays of function pointers are generated which are then linked into the program. The (static) linker is responsible for including the library functions in the interpreter's address space and placing a reference to them inside these arrays. The Python callable objects generated for each library function include indices into these arrays of function pointers rather than direct references to the functions themselves, eliminating the need for any run-time linking to determine the function address.

When a Python program calls one of these foreign functions, the interpreter first references the arrays of function pointers to find the address of the function to call. Since the function was already loaded into the interpreter's address space by the compiler, there is no run-time library load process like there is in the desktop `libffi` implementation. From here, arguments are converted automatically from Python objects to C variables and the address of the function is loaded into the program counter, as in the desktop version.

5 Profiling and Analysis

There are many trade-offs in the design of an embedded run-time system. It is critical to measure the characteristics of both programs and the run-time system itself in order to better understand these trade-offs. Owl is the first embedded run-time system to provide a rich suite of performance and memory analyzers that offer insight into these design issues. With very few exceptions, space is more critical than performance for the studied embedded applications, so Owl is designed to favor memory efficiency over performance.

Building a general-purpose C profiler for microcontrollers is exceptionally difficult because the execution environment can vary so much between systems; there is no common file system and no heap. A profiler must be customized for each particular system. For example, the commercial toolchain used for this project (which costs thousands of dollars from a major vendor) leaves much of the provided `gprof` implementation incomplete. The user needs to write assembly code to be called during every function call, somehow recording data from registers into a file on the host for post-processing.

In contrast, building a profiler as a part of a managed run-time system is much easier. Since the interpreter indirectly executes the program one step at a time, it can easily be modified to record information about that execution. This information can be recorded in scratchpad regions of memory, or even inside of other objects on the heap. Since the virtual machine has complete control over the memory space, this is completely transparent to the user's running program.

The Owl run-time system includes a *line number profiler* and a *call trace profiler* that can be turned on and off by the programmer. These are statistical profilers that operate similarly to `gprof` and provide information about the time spent on individual Python lines or functions. It also includes two profilers that measure the performance of the virtual machine itself. Furthermore, the Owl run-time system also includes a memory analyzer, which would not even be possible for conventional C programs. Additionally, the Owl toolchain includes a novel *static binary analyzer* to visualize which portions of the virtual machine take up the most space in flash.

These profilers are useful not only for writing high-performance applications but also for tuning the virtual machine itself. Furthermore, they demonstrate that building tools for run-time analysis is straightforward with an embedded interpreter. This comprehensive suite of measurement tools is unique to Owl; eLua and p14p have no built-in profilers.

All of these profilers operate transparently to the user and store all data directly within the Python heap. This works even when the microcontroller is disconnected from the host. Therefore, these profilers can be used in mobile, untethered systems, which is not possible with any other microcontroller profiler.

5.1 Memory Analyzer

A managed run-time system imposes structure on the memory within the system. Everything in the heap is a Python object with an object descriptor that includes its type and size. Furthermore, all data is stored within the Python heap, including stack frames and other global

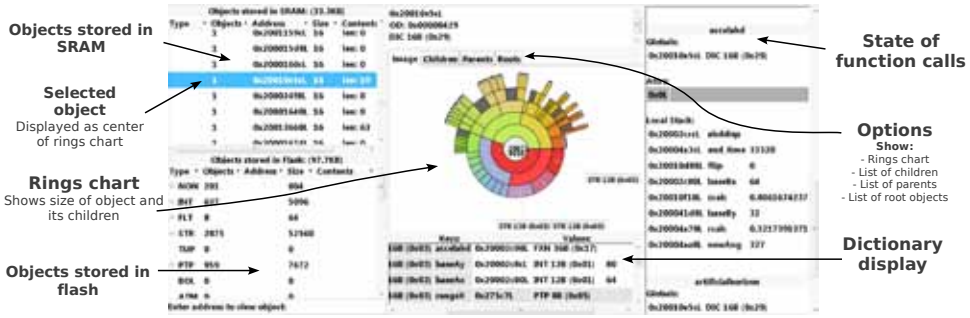


Figure 3: Owl memory analyzer.

and local data. Therefore, it is possible to build tools that can analyze the entire memory space of a Python program. Such tools would actually be impossible to build for C applications, due to the existence of pointers, the fact that data is scattered through the address space, and the lack of a well-defined structure to objects in memory.

The Owl system includes a novel memory analyzer, shown in Figure 3, that provides the programmer with complete access to the entire memory space on the microcontroller. No other system has such a capability. This is accomplished by transferring the entire contents of the Python heap (which is only tens of KB on a Cortex-M3 microcontroller) to the host. The host can then parse the heap and present information about all Python objects that have not yet been garbage collected. Objects in flash that are referenced from the heap can be requested and transferred as well.

The Owl memory analyzer operates by building and traversing a directed graph of the memory objects stored in both the heap and flash. Two tables, one for objects stored in SRAM and one for those in flash, display all objects organized by type. An object can be selected from these tables or looked up by address in order to display additional information about that object, both textually and graphically. Additionally, the memory analyzer can trace the stack frames to show the scope of the objects. These features allow the programmer to easily view how objects are stored, see which objects exist at a given point in the programs execution, and identify costly objects.

5.2 Static Binary Analyzer

The on-chip flash memory of a microcontroller constrains the complexity of the programs and data that the microcontroller can utilize. It is critical to use this scarce resource efficiently. Despite this, most embedded toolchains provide little feedback on flash utilization. The Owl static binary analyzer visualizes the size of the virtual machine in flash, broken down by which portions of the source tree use the most space. An example of its output is shown in Figure 5.

It uses the `nm` tool to extract the list of symbols from the compiled and linked binary. Then, it generates a graphical output showing the size of different parts of the system. Each C source file is shown as a band whose size is proportional to the space that code consumes in the final binary. The bands are sorted by category or directory in the source tree. Larger files are annotated with their file name. This tool was invaluable in identifying sections of the virtual machine that were unnecessarily large, such as newlib's `printf` suite and wrapped functions.

5.3 Python and VM Profilers

The three profilers that measure execution time—the Python line number profiler, the call-graph profiler and the VM bytecode profiler—work in fundamentally similar fashions. A hardware timer fires periodically, stopping the interpreter. The profiler determines which part of the user code the interpreter is currently running. This information is then recorded in a set of counters inside the Python heap. In the case of the line number and call-graph profiler, the profiler records the location of execution inside the user's program. In the VM bytecode profiler, the currently executing bytecode is recorded. After the profiler is turned off, the host can read these records and present them in a human-readable format.

The Python line number and call-graph profilers are useful for writing high-performance Python code. However, the VM profiler is extremely useful in determining bottlenecks in the virtual machine itself. For instance, it revealed that a significant portion of the execution time of most programs was spent looking up variable names.

Since Python is dynamically linked, the VM stores variable names as strings and maintains a dictionary mapping those strings to their values for each namespace. Every time a variable is accessed the interpreter searches this dictionary. This can take a long time since multiple variable names may be looked up during a single line of execution. Additionally, the lookup currently uses a linear scan, so it is inefficient. A dedicated profiler that measures the performance of dictionary lookups quantifies this inefficiency, as discussed in Section 7.

```
for location in self.route:
  ...
  atTarget = False
  while not atTarget:
    ...
    if curtime > range_update_time:
      ... # Read range finder;
          # get distance to nearest obstacle
      if dist_to_obstacle < RANGE_MAX:
        ... # Set motor proportional and servo
            # inversely proportional to distance
      ...
      range_update_time = curtime + RANGE_PERIOD

    if curtime > loc_update_time:
      ... # Read GPS; get current location,
          # heading, and distance to destination
      if dist_to_goal < ERROR_MARGIN:
        ... # Set motor to min speed
        atTarget = True
      else:
        ... # Set motor proportional to dist_to_goal
      loc_update_time = curtime + LOC_PERIOD

    if curtime > heading_update_time:
      ... # Calculate degrees to turn (deg_to_turn)
      heading_update_time = curtime + HEADING_PERIOD

    if curtime > gyro_update_time:
      ... # Read gyro; update steering and integral
      if dist_to_obstacle == RANGE_MAX:
        ... # Set steering to gyro's recommendation
      gyro_update_time = curtime + GYRO_PERIOD

... # Stop the car when it reaches the final location
```

Figure 4: Python event loop for autonomous RC car.

6 Applications

This paper demonstrates the Owl system on the Texas Instruments Stellaris LM3S9B92 (9B92), an ARM Cortex-M3 microcontroller that operates at up to 80 MHz, has 96 KB of SRAM, and has 256 KB of flash. In the experiments, the 9B92 is connected to a GPS receiver, three-axis accelerometer, three-axis MEMS gyroscope, digital compass, TFT display, microSD card reader, ultrasonic range finder, steering servo, and motor controller. The applications use these devices to implement an artificial horizon display (using the display and accelerometer), a GPS tracker (using the GPS, compass, microSD, and display), and an autonomous RC car (using the gyroscope, GPS, range finder, steering servo, and motor controller).

The diversity of peripherals demonstrates the ease of use of a high-level language for microcontroller development. In general, figuring out how to initialize and utilize such peripherals with a microcontroller is a long and tedious process. With Python, however, the ability to experiment within the interactive prompt often shortens the process from days to less than an hour.

In all experiments, the only native C code outside of the run-time system is the StellarisWare libraries; no other foreign functions were utilized. Specifically, no application code has been rewritten in C for performance optimization. Only the profilers discussed in Section 5 were used to measure and improve performance.

6.1 Autonomous RC Car

The autonomous RC car demonstrates the capability, flexibility, and ease of use of the Owl system. The electronics from an off-the-shelf RC car (the Exceed RC Electric SunFire Off-Road Buggy) were replaced with a 9B92 microcontroller and associated peripherals. The car is controlled entirely by the microcontroller.

The code skeleton in Figure 4 shows the main event loop running on the car to implement a feedback controller that performs GPS-based path-following with obstacle avoidance. An ultrasonic range finder, GPS receiver, and three-axis gyroscope connected to the microcontroller transmit feedback from the car's surroundings, while connections to the car's motor and steering servo provide control of the car's movements.

First, if the range finder detects an obstacle in lines 7–8 of the code, the controller translates the distance to that obstacle into a proportional motor speed and an inversely proportional steering servo setting. Second, the controller uses the GPS location to calculate the distance to its destination in lines 16–17. If the car has reached its destination, the controller slows the motor and updates to the next destination. Otherwise, it scales the motor's speed proportionally to the distance and calculates a goal heading. Finally, the a proportional-integral controller steers the car, using the gyro's feedback to control the rate of turn and the GPS' heading calculation to determine the degrees to turn.

The range finder can update every 49 ms, the gyro every 10 ms, and the GPS every second; the controller utilizes these sensors fully by requesting updates every 60, 30, and 1000 ms, respectively.

6.2 Microbenchmarks

A small subset of the Computer Language Shootout[5] shows the computational performance of the system. These benchmarks were ported to the Owl system simply by converting Python 3 code into Python 2 code and removing the command-line arguments.

`ackermann` is a simple, eight line implementation of Ackermann's function that computes $A(3,4)$. It exercises the recursive function call and compute stack. `heapsort` sorts a 1000-element array of random floating point numbers. This implementation is contained in a single function call and exercises the garbage collector and list capabilities. `matrix` multiplies a pair of 30x30 matrices of integers using an $O(n^3)$ algorithm. Finally, `nbody` is

[5] `http://dada.perl.it/shootout/craps.html`
`http://shootout.alioth.debian.org/`

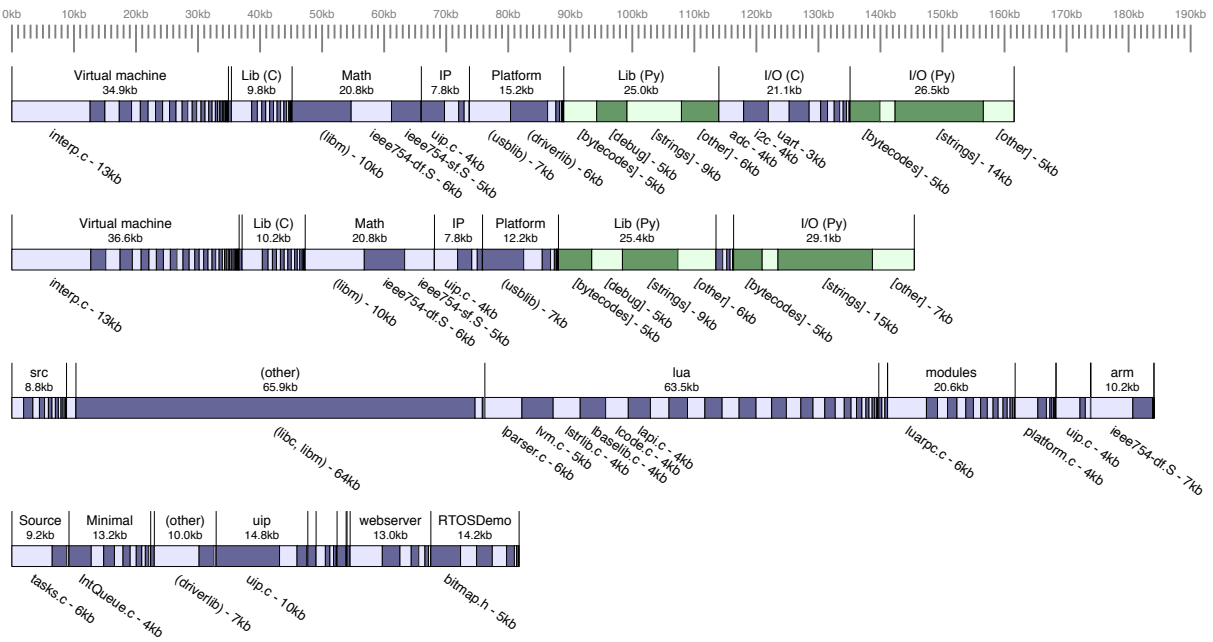


Figure 5: Static binary analysis, from top to bottom, of the Owl virtual machine using wrapped functions, using the foreign function interface, of the eLua virtual machine and of an example program using SafeRTOS.

a floating-point benchmark that simulates the motion of the Sun and the four planets of the outer solar system.

7 Results

This section presents an analysis of the Owl system. The overhead of including the virtual machine in flash can be quite small, as low as 32 KB compared to 22 KB for a simple RTOS. We show that for the embedded workloads, garbage collection has almost no impact on runtime performance. Finally, we show that using a loaderless architecture uses four times less SRAM than a traditional system.

7.1 Static binary analysis

Figure 5 shows the output of the static binary analyzer. The four rows in the figure are the Owl run-time system using autowrapping, the Owl run-time system using eFFI, the eLua interpreter, and the SafeRTOS demonstration program. SafeRTOS is an open-source real-time operating system that is representative of the types of runtime systems in use on microcontrollers today.

Consider the Owl run-time system using autowrapping. The virtual machine section includes the interpreter and support code to create and manage Python objects. The math section includes support for software floating point and mathematical functions, while the IP section provides support for Ethernet networking. The platform section is the StellarisWare peripheral and USB driver libraries. The lib sections (C and Python) are the Python standard libraries for the Owl run-time system. Finally, the I/O sections (C and Python) are the calls to the peripheral library, wrapped by the autowrapper tool.

The Python standard libraries consume a significant fraction of the total flash memory required for the Owl run-time system. The capabilities that these libraries provide are mostly optional, and therefore can be removed to save space. However, they provide many useful and convenient functionalities beyond the basic Python bytecodes, such as string manipulation. These sections also include optional debugging information (5 KB).

With eFFI, the binary is roughly 19 KB smaller, illustrating the advantage of using a foreign function interface. While the code required to manually create stack frames and call functions marginally increases the size of the virtual machine and stores some additional information in the Python code objects, it completely eliminates the need for C wrappers.

The Owl virtual machine itself is actually quite small, approximately 35 KB. It contains all of the code necessary for manipulating objects, interpreting bytecodes, managing threads, and calling foreign functions. This is significantly smaller than eLua's core, which takes up 63 KB, and not much larger than the so-called "light weight" SafeRTOS, which requires 22 KB (the Source and Minimal sections). Note also that the supposed compactness of SafeRTOS is deceptive, as it is statically

	Bytecode/Benchmark	artificialhorizon	gps-tracker	car_range	car_gyro	car_range_gyro	car_gps_gyro	car_gps_gyro_range	ackermann	heapsort	matrix	nbody	Running time
Memory	BINARY_SUBSCR	1%	-	-	-	-	-	-	-	9%	11%	4%	7 μs
	LOAD_ATTR	23%	10%	7%	18%	14%	18%	21%	-	5%	-	-	64 μs
	LOAD_CONST	3%	1%	-	-	-	-	-	11%	5%	1%	8%	8 μs
	LOAD_FAST	1%	1%	1%	-	3%	3%	3%	12%	18%	12%	14%	4 μs
	LOAD_GLOBAL	24%	21%	41%	24%	68%	64%	59%	16%	5%	1%	-	75 μs
	STORE_FAST	-	-	-	-	-	-	-	-	2%	3%	6%	4 μs
	STORE_SUBSCR	-	-	-	-	-	-	-	-	2%	-	3%	6 μs
Math	BINARY_ADD	-	-	-	-	-	-	-	3%	2%	-	2%	10 μs
	BINARY_MULTIPLY	-	-	-	-	-	-	-	-	-	3%	16%	11 μs
	BINARY_POWER	-	-	-	-	-	-	-	-	-	-	11%	116 μs
	BINARY_SUBTRACT	-	-	-	-	-	-	-	6%	-	-	3%	10 μs
	INPLACE_ADD	-	-	-	-	-	-	-	-	3%	3%	5%	10 μs
	INPLACE_SUBTRACT	-	-	-	-	-	-	-	-	-	-	3%	11 μs
Control Flow	CALL_FUNCTION	33%	60%	44%	41%	3%	4%	5%	33%	3%	4%	-	148 μs
	COMPARE_OP	4%	-	1%	-	2%	1%	1%	-	7%	-	-	13 μs
	DUP_TOPX	-	-	-	-	-	-	-	-	-	-	3%	4 μs
	FOR_ITER	-	-	-	-	-	-	-	-	-	5%	1%	6 μs
	JUMP_ABSOLUTE	-	-	-	-	1%	1%	-	-	2%	3%	1%	7 μs
	JUMP_IF_FALSE	-	-	1%	-	2%	2%	2%	-	5%	-	-	5 μs
	JUMP_IF_TRUE	-	-	-	-	-	-	-	6%	-	-	-	5 μs
	POP_BLOCK	-	-	-	-	-	-	-	-	-	2%	-	137 μs
	POP_TOP	-	-	-	-	1%	1%	1%	3%	2%	-	-	2 μs
	RETURN_VALUE	-	-	-	-	-	-	-	4%	1%	2%	-	74 μs
	ROT_THREE	-	-	-	-	-	-	-	-	-	-	2%	3 μs
	SETUP_LOOP	-	-	-	-	-	-	-	-	-	1%	-	15 μs
	UNPACK_SEQUENCE	-	-	-	-	-	-	-	-	-	-	7%	12 μs
	(garbage collector)	1%	-	-	-	-	-	11%	-	17%	40%	3%	3-65 ms

Table 1: Fraction of each workload's running time spent in each bytecode and the average execution time of each bytecode. (Bytecodes that occur few or no times are not shown.)

linked directly into the user application. Therefore, with a complex application, more libraries will need to be included and the gap to Owl, which already contains these libraries, will shrink.

The size of the standard Owl distribution is on the order of 150 KB. Much of this size, however, comes from C libraries. Any C application that uses these libraries needs include them, just as Owl does. While the standard Owl distribution includes a large standard library, Owl can just as easily be compiled without unused libraries. Therefore, the space overhead of using Owl can be as low as 35 KB, the size of the interpreter itself.

7.2 Performance

This section presents the performance of the Owl runtime system using the profilers discussed in Section 5.3. Table 1 shows the profiling results of the benchmarks and applications described in Section 6. Each column (before the last column) shows one workload. The autonomous car workload is shown using the GPS, range finder, gyroscope, or some combination thereof. Each entry shows the percentage of the run-time spent executing any given bytecode. If a bytecode is executed less than 1% of the running time of the program, it is shown as a dash. The average run-time of the bytecode is calculated as an average across all executions from all workloads and is shown on the right. For the purposes of this table, the garbage collector is treated as its own bytecode.

For most applications, the single largest contributer to running time is the CALL_FUNCTION bytecode. This bytecode is particularly complex, as it is responsible for creating call frames, instantiating objects, and calling external functions. When a program calls a foreign C function, the CALL_FUNCTION bytecode does not finish until that function completes, so the profiler attributes the foreign function's execution time to CALL_FUNCTION.

For the embedded workloads, loading and storing values takes a large fraction of the execution time. Python stores variables in a set of dictionaries that map a variable's name, a string, to its value. The LOAD_GLOBAL bytecode loads objects (including functions) from the global namespace, and is particularly slow due to the large size of this namespace.

	Lookups	Hit Rate	Search len	Avg size
LOAD_GLOBAL	129232	0.88	25	41.2
LOAD_ATTR	174374	0.71	10.4	17.0

Table 2: Profiler results showing how dictionaries are used by the interpreter.

Specifically, in the artificial horizon workload, nearly half of the running time is spent in the LOAD_ATTR and LOAD_GLOBAL bytecodes. Table 2 shows how the interpreter uses dictionaries in these two bytecodes. When the user references a global variable, the compiler loads a constant representing the string name of the variable, then calls LOAD_GLOBAL. The interpreter searches the local module's scope for that variable. If it is not found there, the interpreter searches the built-in namespace, which mostly contains built-in functions like `max()` or `int()`. In other words, the interpreter may have to search multiple dictionaries per name lookup. However, these dictionaries are reasonably small. As Table 2 shows, each lookup only needs to search an average of 25 entries to find a global variable and 10 entries to find an object attribute. Since the microcontroller has single-cycle memory access, this means that using a less space efficient, faster data structure may not be appropriate.

Owl's garbage collector (GC) is a simple mark-and-sweep collector that occasionally stops execution for a variable period of time. This uncertainty makes Owl unsuitable for *hard* real-time applications. However, in practice, Owl's GC has no significant impact on our *soft* real-time embedded workloads. In these applications, data structures are reasonably simple, and there are only a few small objects (around 1,500 for the artificial horizon benchmark). This means that GC runs very rarely, and only for a short period of time. For the worst case embedded workload, this is never more than 8ms, 11% of the application's running time.

Further reducing the impact of GC on embedded workloads, our virtual machine runs the collector when the system is otherwise idle. For example, all GC invocations for the car workload occur during sleep times. In other words, the garbage collector *never interrupts or slows useful work*. Moreover, while Owl's current garbage collector does not provide hard real-time guarantees, different garbage collectors exist that do [2].

Unsurprisingly, in the CPU benchmarks garbage collection can be a more significant factor. These CPU benchmarks store more complex data structures on the heap, which take a long time to traverse during the mark phase. Additionally, there are a large number of objects (over 7,500) in the heapsort benchmark that take a long time to go through in the sweep phase. Overall, garbage collection can take up to 65 ms and up to 41% of execution time. Similarly, the bytecodes that manipulate objects (BINARY_ADD, etc.) are only significant for the CPU benchmarks. The embedded workloads are dominated by the bytecodes that perform control flow (JUMP_*, COMPARE_OP, etc.).

	Type	Object count	Avg size (bytes)	Total size (bytes)	Fraction of total heap
primitives	None	2	8.0	16	0%
	int	180	12.2	2188	7%
	float	3	12.0	36	0%
	string	29	19.9	576	2%
	bool	2	12.0	24	0%
sequences	tuple	47	15.2	716	2%
	packed tuple	5	8.0	40	0%
	set	1	12.0	12	0%
	seglist	92	16.5	1520	5%
	segment	251	40.0	10048	32%
	list	7	12.0	84	0%
	dict	200	16.3	3252	10%
	xrange	0	0.0	0	0%
OOP	module	22	36.4	800	3%
	class	14	12.0	168	1%
	function	317	36.1	11440	36%
	instance	2	12.0	24	0%
internal	code obj	2	44.0	88	0%
	packed cobj	1	12.0	12	0%
	thread	1	36.0	36	0%
	method	0	0.0	0	0%
	frame	5	94.4	472	1%
	block	2	20.0	40	0%
	(all)	1185	26.7	31592	100%

Table 3: A snapshot of the heap, broken down by object type, for the `artificialhorizon` workload.

Calling I/O functions is relatively fast. A simple microbenchmark that repeatedly calls a basic peripheral I/O function, accumulating the result in a variable, illustrates the overhead of I/O. This loop was calibrated by accumulating a constant into the variable and using this time as a baseline. For functions accessed with a wrapper function, this I/O call takes 11.4 μs. The foreign function interface is more complex, increasing the call time to 20.8 μs. This time increase is significant, but it is outweighed by the savings in flash.

7.3 Memory use

Table 3 shows a snapshot of the contents of the heap for the artificial horizon workload. It contains roughly 1200 objects for a total of 31 KB of data, which is less than half the available space on the 9B92 microcontroller.

In general, the embedded workloads do not need to store a great deal of dynamic data on the heap. The bulk of the space used on the heap consists of references to other constants. A `segment` object is a portions of a list, and a `function` object points to a code object and the variables in its scope. In contrast, the heapsort bench-

mark stores over 7500 dynamic objects, most of which are integers and lists.

In the artificial horizon workload, the objects in SRAM point to 5056 objects in flash, consuming a total of 98 KB. Most of this space is used by code objects which contain bytecodes, constants, and strings. These are all immutable, so the Owl system keeps them in flash, as discussed in Section 3.2. However, other systems, such as p14p and eLua, would have to copy most of this data out of flash and into SRAM, increasing SRAM usage by over a factor of four. This is a critical advantage of design of the Owl toolchain. Program complexity is limited by flash, not by the much more scarce SRAM.

8 Conclusions

This paper has presented the design and implementation of Owl, a powerful and robust embedded Python run-time system. The Owl system demonstrates that it is both possible and practical to build an efficient, embedded run-time system for modern microcontrollers. Furthermore, this paper has shown that it is straightforward to implement complex embedded control software in Python on top of such a run-time system.

This paper has also illustrated several key points about embedded run-time systems through careful analysis of Owl. First, a large fraction of the binary consists of support libraries that would also need to be included in a native C executable. Second, the run-time characteristics of embedded applications are very different from traditional computational workloads. For instance, garbage collector and math performance have much less of an impact on the types of programs that are likely to be run on a microcontroller than they do on data intensive workloads. Instead, the execution speed is limited by efficient variable lookup and function calls. Finally, an embedded control program often uses many more constants than dynamic objects. By keeping these constants in flash, the overall dynamic memory footprint in SRAM of a complex embedded application can be kept relatively small.

Traditionally, microcontrollers are programmed with hand-coded C or auto-generated code from MATLAB. While this paper has presented many of the advantages of a managed run-time system for microcontrollers, it does not present a comparison to more traditional systems. Such comparisons are an important direction for future work to further quantify the trade-offs of embedded managed run-time systems.

There are orders of magnitude more embedded microcontrollers in the world than conventional microprocessors, yet they are much harder to program. This persists because software development for embedded microcontrollers is mired in decades old technology. As a result, there has been a proliferation of low-level embedded software that is difficult to write, difficult to test, and difficult to port to new systems. The Owl system helps to improve this situation by enabling interactive software development in a high-level language on embedded microcontrollers. We believe this will lead to enormous productivity gains that cannot be overstated.

Acknowledgments

We would like to thank Kathleen Foster and Laura Weber for their contributions to the system and applications, Dean Hall for developing p14p, Texas Instruments for hardware donations, and the anonymous reviewers and our shepherd for their advice on the paper.

References

[1] ANH, T. N. B., AND TAN, S.-L. Real-time operating systems for small microcontrollers. *IEEE Micro 29*, 5 (2009).

[2] BACON, D. F., CHENG, P., AND RAJAN, V. T. The metronome: A simpler approach to garbage collection in real-time systems. In *P. 1st JTRES* (2003).

[3] BAYNES, K., COLLINS, C., FILTERMAN, E., ET AL. The performance and energy consumption of embedded real-time operating systems. *IEEE Trans. Computers 52*, 11 (2003).

[4] BEAZLEY, D. M., FLETCHER, D., AND DUMONT, D. Perl extension building with SWIG. In *O'Reilly Perl Conference 2.0* (1998).

[5] BORNSTEIN, D. Dalvik VM internals. In *Google I/O* (2008).

[6] CHEN, Z. *Java Card Technology for Smart Cards*. Addison-Wesley, Boston, MA, USA, 2000.

[7] DATABEANS. Smartphones boost microcontroller shipments... `http://www.mtemag.com/ArticleItem.aspx?Cont_Title=Smartphones+boost+microcontroller+shipments+with+Arm+seeing+major+growth`, March 2, 2011.

[8] GANNSSLE, J. The challenges of real-time programming. *Embedded System Programming Magazine* (2007).

[9] GARTNER, INC. Gartner says pc shipments to slow... `http://www.gartner.com/it/page.jsp?id=1786014`, September 8, 2011.

[10] HILL, J., SZEWCZYK, R., WOO, A., ET AL. System architecture directions for networked sensors. In *P. 9th ASPLOS* (2000).

[11] KALINSKY, D. Basic concepts of real-time operating systems. *LinuxDevices Magazine* (2003).

[12] KUHNEL, C., AND ZAHNERT, K. *BASIC Stamp: An Introduction to Microcontrollers*. Newnes, Woburn, MA, USA, 2000.

[13] LEVIS, P., AND CULLER, D. Maté: a tiny virtual machine for sensor networks. In *P. 10th ASPLOS* (2002).

[14] LIANG, S., AND BRACHA, G. Dynamic class loading in the Java virtual machine. In *P. 13th OOPSLA* (1998).

[15] MCLURKIN, J., LYNCH, A. J., RIXNER, S., ET AL. A low-cost multi-robot system for research, teaching, and outreach. In *10th DARS* (2010).

[16] REES, J., AND HONEYMAN, P. Webcard: a Java card web server. In *P. 4th CARDIS* (2001).

[17] RIPPERT, C., AND HAGIMONT, D. An evaluation of the Java card environment. In *P. Middleware for Mobile Computing* (2001).

AddressSanitizer: A Fast Address Sanity Checker

Konstantin Serebryany, Derek Bruening, Alexander Potapenko, Dmitry Vyukov
Google
{*kcc,bruening,glider,dvyukov*}*@google.com*

Abstract

Memory access bugs, including buffer overflows and uses of freed heap memory, remain a serious problem for programming languages like C and C++. Many memory error detectors exist, but most of them are either slow or detect a limited set of bugs, or both.

This paper presents AddressSanitizer, a new memory error detector. Our tool finds out-of-bounds accesses to heap, stack, and global objects, as well as use-after-free bugs. It employs a specialized memory allocator and code instrumentation that is simple enough to be implemented in any compiler, binary translation system, or even in hardware.

AddressSanitizer achieves efficiency without sacrificing comprehensiveness. Its average slowdown is just 73% yet it accurately detects bugs at the point of occurrence. It has found over 300 previously unknown bugs in the Chromium browser and many bugs in other software.

1 Introduction

Dozens of memory error detection tools are available on the market [3, 8, 11, 13, 15, 21, 23, 24]. These tools differ in speed, memory consumption, types of detectable bugs, probability of detecting a bug, supported platforms, and other characteristics. Many tools succeed in detecting a wide range of bugs but incur high overhead, or incur low overhead but detect fewer bugs. We present AddressSanitizer, a new tool that combines performance and coverage. AddressSanitizer finds out-of-bounds accesses (for heap, stack, and global objects) and uses of freed heap memory at the relatively low cost of 73% slowdown and 3.4x increased memory usage, making it a good choice for testing a wide range of C/C++ applications.

AddressSanitizer consists of two parts: an instrumentation module and a run-time library. The instrumentation module modifies the code to check the *shadow state* for each memory access and creates *poisoned redzones* around stack and global objects to detect overflows and underflows. The current implementation is based on the LLVM [4] compiler infrastructure. The run-time library replaces `malloc`, `free` and related functions, creates poisoned redzones around allocated heap regions, delays the reuse of freed heap regions, and does error reporting.

1.1 Contributions

In this paper we:

- show that a memory error detector can leverage the comprehensiveness of shadow memory with much lower overhead than the conventional wisdom;
- present a novel shadow state encoding that enables compact shadow memory – as much as a 128-to-1 mapping – for detecting out-of-bounds and use-after-free bugs;
- describe a specialized memory allocator targeting our shadow encoding;
- evaluate a new publicly available tool that efficiently identifies memory bugs.

1.2 Outline

After summarizing related work in the next section, we describe the AddressSanitizer algorithm in Section 3. Experimental results with AddressSanitizer are provided in Section 4. We discuss further improvements in Section 5 and then conclude the paper.

2 Related Work

This section explores the range of existing memory detection tools and techniques.

2.1 Shadow Memory

Many different tools use *shadow memory* to store metadata corresponding to each piece of application data. Typically an application address is mapped to a shadow address either by a direct scale and offset, where the full application address space is mapped to a single shadow address space, or by extra levels of translation involving table lookups. Examples of direct mapping include TaintTrace [10] and LIFT [26]. TaintTrace requires a shadow space of equal size to the application address space, which results in difficulties supporting applications that cannot survive with only one-half of their normal address space. The shadow space in LIFT is one-eighth of the application space.

To provide more flexibility in address space layout, some tools use multi-level translation schemes. Valgrind [20] and Dr. Memory [8] split their shadow memory into pieces and use a table lookup to obtain the shadow address, requiring an extra memory load. For 64-bit platforms, Valgrind uses an additional table layer for application addresses not in the lower 32GB.

Umbra [30, 31] combines layout flexibility with efficiency, avoiding a table lookup via a non-uniform and dynamically-tuned scale and offset scheme. BoundLess [9] stores some of its metadata in 16 higher bits of 64-bit pointers, but falls back to more traditional shadow memory on the slow path. LBC [12] performs a fast-path check using special values stored in the application memory and relies on two-level shadow memory on the slow path.

2.2 Instrumentation

A large number of memory error detectors are based on binary instrumentation. Among the most popular are Valgrind (Memcheck) [21], Dr. Memory [8], Purify[13], BoundsChecker [17], Intel Parallel Inspector [15] and Discover [23]. These tools find out-of-bounds and use-after-free bugs for heap memory with (typically) no false positives. To the best of our knowledge none of the tools based on binary instrumentation can find out-of-bounds bugs in the stack (other than beyond the top of the stack) or globals. These tools additionally find uninitialized reads.

Mudflap [11] uses compile-time instrumentation and hence is capable of detecting out-of-bounds accesses for stack objects. However, it does not insert redzones between different stack objects in one stack frame and will thus not detect all stack buffer overflow bugs. It is also known to have false positive reports in complex C++ code[1].

[1] `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=19319`

CCured [19] combines instrumentation with static analysis (only for C programs) to eliminate redundant checks; their instrumentation is incompatible with uninstrumented libraries.

LBC [12] uses source-to-source transformation and relies on CCured to eliminate redundant checks. LBC is limited to the C language and does not handle use-after-free bugs.

Insure++ [24] relies mainly on compile-time instrumentation but also uses binary instrumentation. Details of its implementation are not publicly available.

2.3 Debug Allocators

Another class of memory error detectors uses a specialized memory allocator and does not change the rest of the execution.

Tools like Electric Fence [25], Duma [3], GuardMalloc [16] and Page Heap [18] use CPU page protection. Each allocated region is placed into a dedicated page (or a set of pages). One extra page at the right (and/or at the left) is allocated and marked as inaccessible. A subsequent page fault accessing these pages is then reported as an out-of-bounds error. These tools incur large memory overheads and may be very slow on `malloc`-intensive applications (as each `malloc` call requires at least one system call). Also, these tools may miss some classes of bugs (e.g., reading the byte at offset 6 from the start of a 5-byte memory region). If a bug is reported, the responsible instruction is provided in the error message.

Some other `malloc` implementations, including DieHarder [22] (a descendant of DieHard [5] `malloc`) and Dmalloc [2], find memory bugs on a probabilistic and/or delayed basis. Their modified `malloc` function adds redzones around memory regions returned to the user and populates the newly allocated memory with special *magic* values. The `free` function also writes magic values to the memory region.

If a magic value is read then the program has accessed an out-of-bounds or uninitialized value. However, there is no immediate detection of this. Through properly selected magic values, there is a chance that the program will behave incorrectly in a way detectable by existing application tests (DieHard [5] has a replicated mode in which it is able to detect such incorrect behavior by comparing the output of several program replicas initialized with different magic values). In other words, the detection of out-of-bounds reads and read-after-free bugs is probabilistic.

If a magic value in a redzone is overwritten, this will later be detected when the redzone is examined on `free`, but the tool does not know exactly when the out-of-bounds write or write-after-free occurred. For large programs it is often equivalent to reporting "your program

has a bug". Note that the goal of DieHarder is not only to detect bugs, but also to protect from security attacks.

The two debug `malloc` approaches are often combined. Debug `malloc` tools do not handle stack variables or globals.

The same magic value technique is often used for buffer overflow protection. StackGuard [29] and ProPolice [14] (the StackGuard reimplementation currently used by GCC) place a canary value between the local variables and the return address in the current stack frame and check for that value's consistency upon function exit. This helps to prevent stack smashing buffer overflows, but is unable to detect arbitrary out-of-bounds accesses to stack objects.

3 AddressSanitizer Algorithm

From a high level, our approach to memory error detection is similar to that of the Valgrind-based tool AddrCheck [27]: use shadow memory to record whether each byte of application memory is safe to access, and use instrumentation to check the shadow memory on each application load or store. However, AddressSanitizer uses a more efficient shadow mapping, a more compact shadow encoding, detects errors in stack and global variables in addition to the heap and is an order of magnitude faster than AddrCheck. The following sections describe how AddressSanitizer encodes and maps its shadow memory, inserts its instrumentation, and how its run-time library operates.

3.1 Shadow Memory

The memory addresses returned by the `malloc` function are typically aligned to at least 8 bytes. This leads to the observation that any aligned 8-byte sequence of application heap memory is in one of 9 different states: the first k $(0 \leq k \leq 8)$ bytes are addressable and the remaining $8-k$ bytes are not. This state can be encoded into a single byte of shadow memory.

AddressSanitizer dedicates one-eighth of the virtual address space to its shadow memory and uses a direct mapping with a scale and offset to translate an application address to its corresponding shadow address. Given the application memory address `Addr`, the address of the shadow byte is computed as `(Addr>>3)+Offset`. If `Max-1` is the maximum valid address in the virtual address space, the value of `Offset` should be chosen in such a way that the region from `Offset` to `Offset+Max/8` is not occupied at startup. Unlike in Umbra [31], the `Offset` must be chosen statically for every platform, but we do not see this as a serious limitation. On a typical 32-bit Linux or MacOS system, where the virtual address space is `0x00000000-0xffffffff`, we use `Offset = 0x20000000` (2^{29}). On a 64-bit system with 47 significant address bits we use `Offset = 0x0000100000000000` (2^{44}). In some cases (e.g., with `-fPIE/-pie` compiler flags on Linux) a zero offset can be used to simplify instrumentation even further.

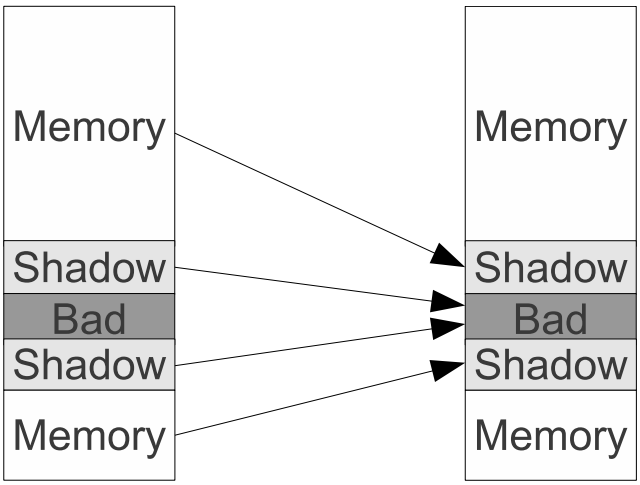


Figure 1: AddressSanitizer memory mapping.

Figure 1 shows the address space layout. The application memory is split into two parts (low and high) which map to the corresponding shadow regions. Applying the shadow mapping to addresses in the shadow region gives us addresses in the *Bad* region, which is marked inaccessible via page protection.

We use the following encoding for each shadow byte: 0 means that all 8 bytes of the corresponding application memory region are addressable; k $(1 \leq k \leq 7)$ means that the first k bytes are addressible; any negative value indicates that the entire 8-byte word is unaddressable. We use different negative values to distinguish between different kinds of unaddressable memory (heap redzones, stack redzones, global redzones, freed memory).

This shadow mapping could be generalized to the form `(Addr>>Scale)+Offset`, where `Scale` is one of $1 \ldots 7$. With `Scale=N`, the shadow memory occupies $1/2^N$ of the virtual address space and the minimum size of the redzone (and the `malloc` alignment) is 2^N bytes. Each shadow byte describes the state of 2^N bytes and encodes 2^N+1 different values. Larger values of `Scale` require less shadow memory but greater redzone sizes to satisfy alignment requirements. Values of `Scale` greater than 3 require more complex instrumentation for 8-byte accesses (see Section 3.2) but provide more flexibility with applications that may not be able to give up a single contiguous one-eighth of their address space.

3.2 Instrumentation

When instrumenting an 8-byte memory access, AddressSanitizer computes the address of the corresponding shadow byte, loads that byte, and checks whether it is zero:

```
ShadowAddr = (Addr >> 3) + Offset;
if (*ShadowAddr != 0)
  ReportAndCrash(Addr);
```

When instrumenting 1-, 2-, or 4- byte accesses, the instrumentation is slightly more complex: if the shadow value is positive (i.e., only the first k bytes in the 8-byte word are addressable) we need to compare the 3 last bits of the address with k.

```
ShadowAddr = (Addr >> 3) + Offset;
k = *ShadowAddr;
if (k != 0 && ((Addr & 7) + AccessSize > k))
    ReportAndCrash(Addr);
```

In both cases the instrumentation inserts only one memory read for each memory access in the original code. We assume that an N-byte access is aligned to N. AddressSanitizer may miss a bug caused by an unaligned access, as described in Section 3.5.

We placed the AddressSanitizer instrumentation pass at the very end of the LLVM optimization pipeline. This way we instrument only those memory accesses that survived all scalar and loop optimizations performed by the LLVM optimizer. For example, memory accesses to local stack objects that are optimized away by LLVM will not be instrumented. At the same time we don't have to instrument memory accesses generated by the LLVM code generator (e.g., register spills).

The error reporting code (`ReportAndCrash(Addr)`) is executed at most once, but is inserted in many places in the code, so it still must be compact. Currently we use a simple function call (see examples in Appendix A). Another option would be to use an instruction that generates a hardware exception.

3.3 Run-time Library

The main purpose of the run-time library is to manage the shadow memory. At application startup the entire shadow region is mapped so that no other part of the program can use it. The *Bad* segment of the shadow memory is protected. On Linux the shadow region is always unoccupied at startup so the memory mapping always succeeds. On MacOS we need to disable the address space layout randomization (ASLR). Our preliminary experiments show that the same shadow memory layout also works on Windows.

The `malloc` and `free` functions are replaced with a specialized implementation. The `malloc` function allocates extra memory, the redzone, around the returned region. The redzones are marked as unaddressable, or *poisoned*. The larger the redzone, the larger the overflows or underflows that will be detected.

The memory regions inside the allocator are organized as an array of freelists corresponding to a range of object sizes. When a freelist that corresponds to a requested object size is empty, a large group of memory regions with their redzones is allocated from the operating system (using, e.g., `mmap`). For n regions we allocate $n+1$ redzones, such that the right redzone of one region is typically a left redzone of another region:

rz1	mem1	rz2	mem2	rz3	mem3	rz4

The left redzone is used to store the internal data of the allocator (such as the allocation size, thread ID, etc.); consequently, the minimum size of the heap redzone is currently 32 bytes. This internal data can not be corrupted by a buffer underflow, because such underflows are detected immediately prior to the actual store (if the underflow happens in the instrumented code).

The `free` function poisons the entire memory region and puts it into *quarantine*, such that this region will not be allocated by `malloc` any time soon. Currently, the quarantine is implemented as a FIFO queue which holds a fixed amount of memory at any time.

By default, `malloc` and `free` record the current call stack in order to provide more informative bug reports. The `malloc` call stack is stored in the left redzone (the larger the redzone, the larger the number of frames that can be stored) while the `free` call stack is stored in the beginning of the memory region itself.

Section 4.3 discusses how to tune the run-time library.

3.4 Stack And Globals

In order to detect out-of-bounds accesses to globals and stack objects, AddressSanitizer must create poisoned redzones around such objects.

For globals, the redzones are created at compile time and the addresses of the redzones are passed to the run-time library at application startup. The run-time library function poisons the redzones and records the addresses for further error reporting.

For stack objects, the redzones are created and poisoned at run-time. Currently, redzones of 32 bytes (plus up to 31 bytes for alignment) are used. For example, given a program

```
void foo() {
  char a[10];
  <function body> }
```

the transformed code will look like

```
void foo() {
  char rz1[32]
  char arr[10];
  char rz2[32-10+32];
  unsigned *shadow =
    (unsigned*)(((long)rz1>>8)+Offset);
  // poison the redzones around arr.
  shadow[0] = 0xffffffff;  // rz1
  shadow[1] = 0xffff0200;  // arr and rz2
  shadow[2] = 0xffffffff;  // rz2
  <function body>
  // un-poison all.
  shadow[0] = shadow[1] = shadow[2] = 0;  }
```

3.5 False Negatives

The instrumentation scheme described above may miss a very rare type of bug: an unaligned access that is partially out-of-bounds. For example:

```
int *a = new int[2]; // 8-aligned
int *u = (int*)((char*)a + 6);
*u = 1;  // Access to range [6-9]
```

Currently we ignore this type of bug since all solutions we have come up with slow down the common path. Solutions we considered include:

- check at run-time whether the address is unaligned;
- use a byte-to-byte shadow mapping (feasible only on 64-bit systems);
- use a more compact mapping (e.g., `Scale=7` from Section 3.1) to minimize the probability of missing such a bug.

AddressSanitizer may also miss bugs in the following two cases (tools like Valgrind or Dr. Memory have the same problems). First, if an out-of-bounds access touches memory too far away from the object bound it may land in a different valid allocation and the bug will be missed.

```
char *a = new char[100];
char *b = new char[1000];
a[500] = 0;  // may end up somewhere in b
```

All out-of-bounds accesses within the heap redzone will be detected with 100% probability. If the memory footprint is not a serious constraint we recommend using large redzones of up to 128 bytes.

Second, a use-after-free may not be detected if a large amount of memory has been allocated and deallocated between the "free" and the following use.

```
char *a = new char[1 << 20];  // 1MB
delete [] a;  // <<< "free"
char *b = new char[1 << 28];  // 256MB
delete [] b;  // drains the quarantine queue.
char *c = new char[1 << 20];  // 1MB
a[0] = 0;     // "use". May land in 'c'.
```

3.6 False Positives

In short, AddressSanitizer has no false positives. However, during the AddressSanitizer development and deployment we have seen a number of undesirable error reports described below, all of which are now fixed.

3.6.1 Conflict With Load Widening

A very common compiler optimization called *load widening* conflicts with AddressSanitizer instrumentation. Consider the following C code:

```
struct X { char a, b, c; };
void foo() {
  X x; ...
  ... = x.a + x.c; }
```

In this code, the object `x` has size 3 and alignment 4 (at least). Load widening transforms `x.a+x.c` into one 4-byte load, which partially crosses the object boundary. Later in the optimization pipeline AddressSanitizer instruments this 4-byte load which leads to a false positive. To avoid this problem we partially disabled load widening in LLVM when AddressSanitizer instrumentation is enabled. We still allow widening `x.a+x.b` into a 2-byte load, because such a transformation will not cause false positives and will speedup the instrumented code.

3.6.2 Conflict With Clone

We have observed several false reports in the presence of the `clone` system call[2]. First, a process calls `clone` with the `CLONE_VM|CLONE_FILES` flags, which creates a child process that shares memory with the parent. In particular, the memory used by the child's stack still belongs to the parent. Then the child process calls a function that has objects on the stack and the AddressSanitizer instrumentation poisons the stack object redzones. Finally, without exiting the function and un-poisoning the redzones, the child process calls a function that never returns (e.g., `_exit` or `exec`). As a result, part of the parent address space remains poisoned and AddressSanitizer reports an error later when this memory is reused. We solved this problem by finding `never_return` function calls (functions like `_exit` or `exec` have this attribute) and unpoisoning the entire stack memory before the call. For

[2] http://code.google.com/p/address-sanitizer/issues/detail?id=37

similar reasons the AddressSanitizer run-time library has to intercept `longjmp` and C++ exceptions.

3.6.3 Intentional Wild Dereferences

We have seen several cases where a function intentionally reads *wild* memory locations. For example, low level code iterates between two addresses on the stack crossing multiple stack frames. For these cases we have implemented a function attribute `no_address_safety_analysis` which should be added to the function declaration in the C/C++ source. These cases are rare; e.g., in the Chromium browser we needed this attribute only once.

3.7 Threads

AddressSanitizer is thread-safe. The shadow memory is modified only when the corresponding application memory is not accessible (inside `malloc` or `free`, during creation or destruction of a stack frame, during module initialization). All other accesses to the shadow memory are reads. The `malloc` and `free` functions use thread-local caches to avoid locking on every call (as most modern `malloc` implementations do). If the original program has a race between a memory access and deletion of that memory, AddressSanitizer may sometimes detect it as a use-after-free bug, but is not guaranteed to. Thread IDs are recorded for every `malloc` and `free` and are reported in error messages together with thread creation call stacks.

4 Evaluation

We measured the performance of AddressSanitizer on C and C++ benchmarks from SPEC CPU2006 [28]. The measurements were done in 64-bit mode on an HP Z600 machine with 2 quad-core Intel Xeon E5620 CPUs and 24GB RAM. We compared the performance of instrumented binaries with the binaries built using the regular LLVM compiler (`clang -O2`). We used 32-byte redzones, disabled the stack unwinding during `malloc` and `free` and set the quarantine size to zero (see Section 4.3).

Figure 2 shows that the average slowdown on CPU2006 is 73%. The largest slowdown is seen on perlbench and xalancbmk (2.60x and 2.67x respectively). These two benchmarks are very `malloc`-intensive and make a huge number of 1- and 2- byte memory accesses (both benchmarks are text processing programs). We also measured AddressSanitizer performance when only writes are instrumented: the average slowdown is 26%. This mode could be used in performance-critical environments to find a subset of memory bugs.

Table 1: Memory usage with AddressSanitizer (MB)

Benchmark	Original	Instrumented	Increase
400.perlbench	670	2168	3.64x
401.bzip2	858	1618	2.12x
403.gcc	893	4133	5.21x
429.mcf	1684	2098	1.40x
445.gobmk	37	369	11.22x
456.hmmer	33	582	19.84x
458.sjeng	180	249	1.56x
462.libquantum	104	930	10.06x
464.h264ref	72	439	6.86x
471.omnetpp	181	787	4.89x
473.astar	343	1214	3.98x
483.xalancbmk	434	1688	4.38x
433.milc	694	1618	2.62x
444.namd	58	146	2.83x
447.dealII	807	2602	3.63x
450.soplex	637	2479	4.38x
453.povray	17	371	24.55x
470.lbm	417	550	1.48x
482.sphinx3	52	426	9.22x
total	8171	24467	**3.37**x

Three bugs were found in CPU2006: one stack and one global buffer overflow in h264ref, and a use-after-realloc in perlbench.

We also evaluated the performance of different mapping `Scale` and `Offset` values (see Section 3.1). The `Scale` values greater than 3 produce slightly slower code on average (from 2% speedup to 15% slowdown compared to `Scale=3`). The memory footprint for `Scale=4,5` is close to that of `Scale=3`. For values 6 and 7, the memory footprint is greater because larger redzones are required. Setting `Offset` to zero (which requires `-fPIE/-pie`) gives a small speedup, bringing the average slowdown on CPU2006 to 69%.

Table 1 summarizes the increase in memory usage (collected by reading the VmPeak field from `/proc/self/status` at process termination). The memory overhead comes mostly from the `malloc` redzones. The average memory usage increase is 3.37x. There is also a constant-size overhead for quarantine, which we did not count in the experiment.

Table 2 summarizes the stack size increase (`VmStk` field in `/proc/self/status`). Only 6 benchmarks had a noticeable stack size change and only 3 benchmarks had stack size increases over 10%.

The binary size increase on SPEC CPU2006 ranges from 1.5x to 3.2x, with an average of 2.5x.

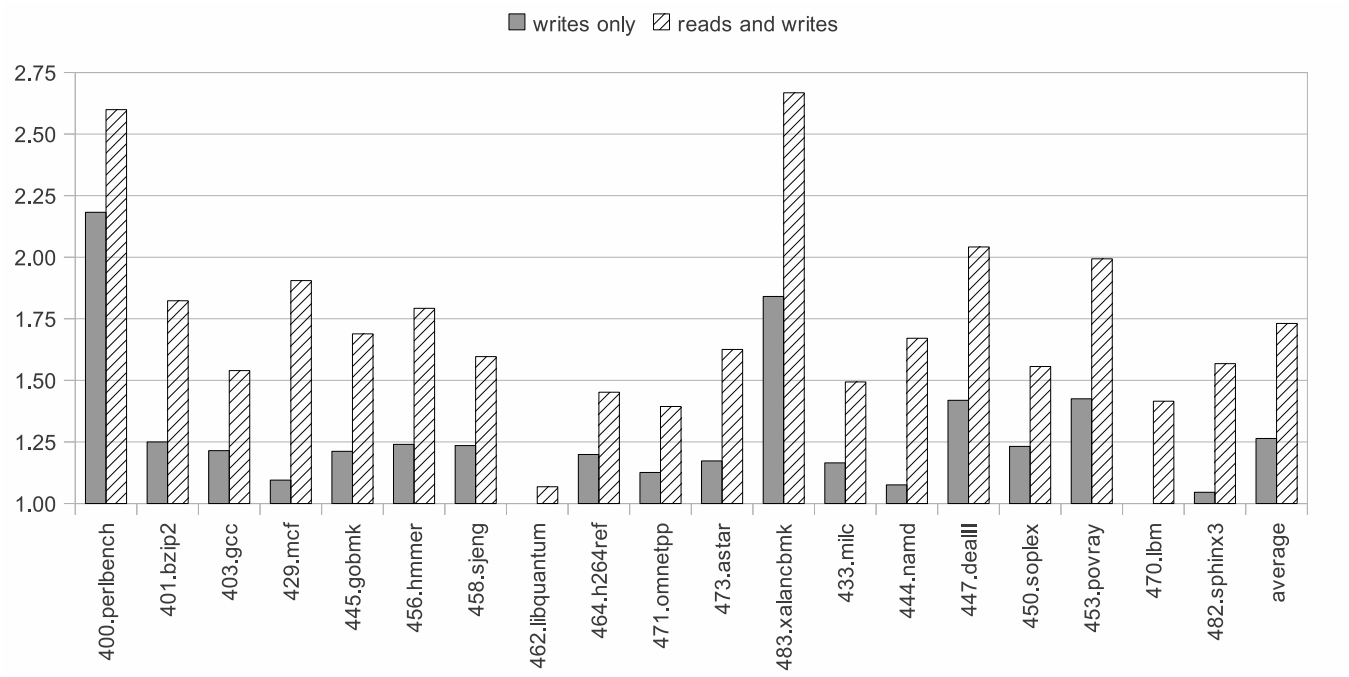


Figure 2: The average slowdown on SPEC CPU2006 on 64-bit Linux.

Table 2: Stack increase with AddressSanitizer (KB)

Benchmark	Original	Instrumented	Increase
400.perlbench	568	1740	3.06x
445.gobmk	184	264	1.43x
458.sjeng	828	848	1.02x
483.xalancbmk	2116	4720	2.23x
453.povray	88	96	1.09x
482.sphinx3	248	252	1.02x

4.1 Comparison

Comparing AddressSanitizer with other tools is tricky because other tools find different sets of bugs. Valgrind and Dr. Memory incur 20x and 10x slowdowns on CPU2006, respectively [8]. But these tools detect a different set of bugs (in addition to out-of-bounds and use-after-free they detect uninitialized reads and memory leaks, but do not handle out-of-bounds for most stack variables and globals).

Mudflap, probably the most similar tool to AddressSanitizer, has very unusual performance characteristics. According to our measurements, Mudflap's slowdown on CPU2006 ranges from 2x to 41x; several benchmarks fail with out-of-memory errors.

Debug malloc implementations that use CPU guard pages typically slow down only very `malloc`-intensive applications. Duma, the freely available guard page implementation for Linux, crashed on 12 out of 18 CPU2006 benchmarks with out-of-memory errors. No surprise: the Duma manual describes it as a "terrible memory hog". On the remaining 6 benchmarks it showed very small overhead (from -1% to 5%).

The overhead of the DieHarder debug malloc is very low, 20% on average [22]. However, on three `malloc`-intensive benchmarks it is comparable with the overhead of AddressSanitizer: perlbench, 2x; omnetpp, 1.85x; xalancbmk, 1.75x.

4.2 AddressSanitizer Deployment

The Chromium open-source browser [1] has been regularly tested with AddressSanitizer since we released the tool in May 2011. In the first 10 months of testing the tool detected over 300 previously unknown bugs in the Chromium code and in third-party libraries. 210 bugs were *heap-use-after-free*, 73 were *heap-buffer-overflow*, 8 *global-buffer-overflow*, 7 *stack-buffer-overflow* and 1 `memcpy` parameter overlap. In 13 more cases AddressSanitizer triggered some other kind of program error (e.g., uninitialized memory read), but did not provide a meaningful error message.

The two major sources of bug reports in Chromium are regular runs of the existing unit tests and targeted random test generation (fuzzing). In either case the speed of the instrumented code is critical. For unit tests, the high speed allows the use of fewer machines to keep up with the source changes. For fuzzing, it allows running

a random test in just a few seconds (since AddressSanitizer is implemented as compile-time instrumentation, it has no start-up penalty), and once a bug is found, minimize the test in reasonable time. A small number of bugs have been found by manually running the instrumented browser — which would not have been possible with a significantly slower tool.

Besides Chromium we have also tested a large amount of other code and found many bugs. As in Chromium, heap-use-after-free was the most frequent kind of bug; however stack- and global-buffer-overflow appeared more often than in Chromium. Several heap-use-after-free bugs were detected in LLVM itself. We were notified about bugs found by AddressSanitizer in Firefox, Perl, Vim, and several other opensource projects[3].

4.3 Tuning Accuracy And Resource Usage

AddressSanitizer has three major controls that affect accuracy and resource usage.

Depth of stack unwinding (default: 30). On every call to `malloc` and `free` the tool needs to unwind the call stack so that error messages contain more information. This option affects the speed of the tool, especially if the tested application is `malloc`-intensive. It does not affect the memory footprint or the bug-finding ability, but short stack traces are often not enough to analyze an error message.

Quarantine size (default: 256MB). This value controls the ability to find heap-use-after-free bugs (see Section 3.5). It does not affect performance.

Size of the heap redzone (default: 128 bytes). This option affects the ability to find heap-buffer-overflow bugs (see Section 3.5). Large values may lead to significant slowdown and increased memory usage, especially if the tested program allocates many small chunks of heap memory. Since the redzone is used to store the `malloc` call stack, decreasing the redzone automatically decreases the maximal unwinding depth.

While testing Chromium we used the default values of these three parameters. Increasing any of them did not increase the bug-finding ability. While testing other software we sometimes had to use smaller redzone sizes (32 or 64 bytes) and/or completely disable stack unwinding to meet extreme memory and speed constraints. In environments with a small amount of RAM we used smaller quarantine sizes. All three values are controlled by an environment variable and can be set at process startup.

[3] http://code.google.com/p/address-sanitizer/wiki/FoundBugs

5 Future Work

This section discusses improvements and further steps that could be taken with AddressSanitizer.

5.1 Compile-time Optimizations

It is not necessary to instrument every memory access in order to find all memory bugs. There is redundant instrumentation that can be eliminated, as shown in this example:

```
void inc(int *a) {
  (*a)++;  }
```

Here we have two memory accesses, one load and one store, but we need to instrument only the first one. This is the only compile-time optimization implemented in AddressSanitizer currently. Some other possible optimizations are described below. These optimizations apply only under certain conditions (e.g., there should be no non-pure function calls between the two accesses in the first example).

• Instrument only the first access:

```
*a = ...
if (...)
  *a = ...
```

• Instrument only the second access (although this gives up the property of guaranteeing to report an error prior to the actual load or store taking place):

```
if (...)
  *a = ...
*a = ...
```

• Instrument only `a[0]` and `a[n-1]`:

```
for (int i = 0; i < n; i++)
  a[i] = ...;
```

We already use this approach for instrumenting functions like `memset`, `memcpy` and similar. It may potentially miss some bugs if `n` is large.

• Combine two accesses into one:

```
struct { int a, b; } x;  ...
x.a = ...;
x.b = ...;
```

• Do not instrument accesses that can be statically proven to be correct:

```
int x[100];
for (int i = 0; i < 100; i++)
  x[i] = ...;
```

• No point in instrumenting accesses to scalar globals:

```
int glob;
int get_glob() {
  return glob; }
```

5.2 Handling Libraries

The current implementation of AddressSanitizer is based on compile-time instrumentation and thus does not handle system libraries (it does, however, handle some C library functions such as `memset`). For the open source libraries the best approach might be to create special instrumented builds. For the closed source libraries a combined static/dynamic instrumentation approach could be used. All available source code could be built with an AddressSanitizer-enabled compiler. Then, during execution, the closed-source libraries could be instrumented with a binary translation system (such as DynamoRIO [7, 6]).

It is possible to implement AddressSanitizer using only run-time instrumentation but it will probably be slower due to binary translation overhead, including suboptimal register allocation. Besides, it is not clear how to implement redzones for stack objects using run-time instrumentation.

5.3 Hardware Support

The performance characteristics of AddressSanitizer allow for use in a wide range of situations. However, for the most performance-critical applications and for the cases where the binary size is important, the current overhead may be too restrictive. The instrumentation performed by AddressSanitizer (see Section 3.2) could be replaced by a single new hardware instruction `checkN` (e.g., "`check4 Addr`" for a 4-byte access). The `checkN` instruction with parameter `Addr` should be equivalent to

```
ShadowAddr = (Addr >> Scale) + Offset;
k = *ShadowAddr;
if (k != 0 && ((Addr & 7) + N > k)
    GenerateException();
```

The values of `Offset` and `Scale` could be stored in special registers and set at application startup.

Such an instruction would improve performance by reducing the icache pressure, combining simple arithmetic operations, and achieving better branch prediction. It would also reduce the binary size significantly.

By default the `checkN` instruction could be a no-op and only enabled by a special CPU flag. This would allow to selectively test certain executions or even test long-lived processes for a fraction of their execution time.

6 Conclusions

In this paper we presented AddressSanitizer, a fast memory error detector. AddressSanitizer finds out-of-bounds (for heap, stack, and globals) accesses and use-after-free bugs at the cost of 73% slowdown on average; the tool has no false positives.

AddressSanitizer uses shadow memory to provide accurate and immediate bug detection. The conventional wisdom is that shadow memory either incurs high overhead through multi-level mapping schemes or imposes prohibitive address space requirements by occupying a large contiguous region. Our novel shadow state encoding reduces our shadow space footprint enough that we can use a simple mapping, which can be implemented with low overhead.

The high speed provided by the tool allows the user to run more tests faster. The tool has been used to test the Chromium browser and has found over 300 real bugs in just 10 months, including some that could have potentially led to security vulnerabilities. AddressSanitizer users found bugs in Firefox, Perl, Vim and LLVM.

The instrumentation required for AddressSanitizer is simple enough to be implemented in a wide range of compilers, binary instrumentation systems, and even in hardware.

Availability

AddressSanitizer is open source and is integrated with the LLVM compiler tool chain [4] starting from version 3.1. The documentation can be found at `http://clang.llvm.org/docs/AddressSanitizer.html`.

A Appendix: Instrumentation Examples

Here we give two examples of instrumentation on x86_64 (8- and 4- byte stores). C program:

```
void foo(T *a) {
  *a = 0x1234;
}
```

8-byte store:

```
clang -O2 -faddress-sanitizer a.c -c -DT=long

push   %rax
mov    %rdi,%rax
shr    $0x3,%rax
mov    $0x100000000000,%rcx
or     %rax,%rcx
cmpb   $0x0,(%rcx) # Compare Shadow with 0
jne    23 <foo+0x23>  # To Error
movq   $0x1234,(%rdi) # Original store
pop    %rax
retq
callq  __asan_report_store8 # Error
```

4-byte store:

```
clang -O2 -faddress-sanitizer a.c -c -DT=int
```

```
push   %rax
mov    %rdi,%rax
shr    $0x3,%rax
mov    $0x100000000000,%rcx
or     %rax,%rcx
mov    (%rcx),%al  # Get Shadow
test   %al,%al
je     27 <foo+0x27> # To original store
mov    %edi,%ecx  # Slow path
and    $0x7,%ecx  # Slow path
add    $0x3,%ecx  # Slow path
cmp    %al,%cl
jge    2f <foo+0x2f> # To Error
movl   $0x1234,(%rdi) # Original store
pop    %rax
retq
callq  __asan_report_store4  # Error
```

References

[1] The Chromium project. http://dev.chromium.org.

[2] Dmalloc – Debug Malloc Library. http://www.dmalloc.com.

[3] D.U.M.A. – Detect Unintended Memory Access. http://duma.sourceforge.net/.

[4] The LLVM Compiler Infrastructure. http://llvm.org.

[5] Emery D. Berger and Benjamin G. Zorn. DieHard: probabilistic memory safety for unsafe languages. In *PLDI 06*, pages 158–168. ACM Press, 2006.

[6] Derek Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, M.I.T., September 2004.

[7] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proc. of the International Symposium on Code Generation and Optimization (CGO '03)*, pages 265–275, March 2003.

[8] Derek Bruening and Qin Zhao. Practical memory checking with Dr. Memory. In *Proc. of the International Symposium on Code Generation and Optimization (CGO '11)*, pages 213–223, April 2011.

[9] Marc Brünink, Martin Süßkraut, and Christof Fetzer. Boundless memory allocations for memory safety and high availability. In *Proc. of the 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2011)*. IEEE Computer Society, June 2011.

[10] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. Taint-trace: Efficient flow tracing with dynamic binary rewriting. In *Proc. of the 11th IEEE Symposium on Computers and Communications (ISCC '06)*, pages 749–754, 2006.

[11] Frank Ch. Eigler. Mudflap: pointer use checking for C/C++. Red Hat Inc.

[12] Niranjan Hasabnis, Ashish Misra, and R. Sekar. Light-weight bounds checking. In *Proc. of the International Symposium on Code Generation and Optimization (CGO '12)*, pages 135–144, April 2012.

[13] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proc. of the Winter USENIX Conference*, pages 125–136, January 1992.

[14] IBM Research. GCC extension for protecting applications from stack-smashing attacks. http://researchweb.watson.ibm.com/trl/projects/security/ssp/.

[15] Intel. Intel Parallel Inspector. http://software.intel.com/en-us/intel-parallel-inspector/.

[16] Mac OS X Developer Library. Memory Usage Performance Guidelines: Enabling the Malloc Debugging Features. http://developer.apple.com/library/mac/#documentation/darwin/reference/manpages/man3/libgmalloc.3.html.

[17] Micro Focus. BoundsChecker. http://www.microfocus.com/products/micro-focus-developer/devpartner/visual-c.aspx.

[18] Microsoft Support. How to use Pageheap.exe in Windows XP, Windows 2000, and Windows Server 2003. http://support.microsoft.com/kb/286470.

[19] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrotting of legacy code. In *Proc. of the , Principles of Programming Languages*, pages 128–139, 2002.

[20] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proc. of the 3rd International Conference on Virtual Execution Environments (VEE '07)*, pages 65–74, June 2007.

[21] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, pages 89–100, June 2007.

[22] Gene Novark and Emery D. Berger. DieHarder: securing the heap. In *Proc. of the 17th ACM conference on Computer and communications security*, CCS '10, pages 573–584. ACM, 2010.

[23] Oracle. Sun Memory Error Discovery Tool (Discover). http://download.oracle.com/docs/cd/E19205-01/821-1784/6nmoc18gq/index.html.

[24] Parasoft. Insure++. http://www.parasoft.com/jsp/products/insure.jsp?itemId=63.

[25] Bruce Perens. Electric Fence. http://perens.com/FreeSoftware/ElectricFence/.

[26] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *Proc. of the 39th International Symposium on Microarchitecture (MICRO 39)*, pages 135–148, 2006.

[27] Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proc. of the USENIX Annual Technical Conference*, pages 2–2, 2005.

[28] Standard Performance Evaluation Corporation. SPEC CPU2006 benchmark suite, 2006. http://www.spec.org/osg/cpu2006/.

[29] Perry Wagle and Crispin Cowan. Stackguard: Simple stack smash protection for gcc. In *Proc. of the GCC Developers Summit*, pages 243–255, 2003.

[30] Qin Zhao, Derek Bruening, and Saman Amarasinghe. Efficient memory shadowing for 64-bit architectures. In *Proc. of the The International Symposium on Memory Management (ISMM '10)*, pages 93–102, Jun 2010.

[31] Qin Zhao, Derek Bruening, and Saman Amarasinghe. Umbra: Efficient and scalable memory shadowing. In *Proc. of the International Symposium on Code Generation and Optimization (CGO '10)*, pages 22–31, April 2010.

Software Persistent Memory

Jorge Guerra, Leonardo Mármol, Daniel Campello, Carlos Crespo, Raju Rangaswami, Jinpeng Wei
Florida International University
{jguerra, lmarm001, dcamp020, ccres008, raju, weijp}@cs.fiu.edu

Abstract

Persistence of in-memory data is necessary for many classes of application and systems software. We propose Software Persistent Memory (SoftPM), a new memory abstraction which allows *malloc* style allocations to be selectively made persistent with relative ease. Particularly, SoftPM's persistent *containers* implement automatic, orthogonal persistence for all in-memory data reachable from a developer-defined *root* structure. Writing new applications or adapting existing applications to use SoftPM only requires identifying such root structures within the code. We evaluated the correctness, ease of use, and performance of SoftPM using a suite of microbenchmarks and real world applications including a distributed MPI application, SQLite (an in-memory database), and memcachedb (a distributed memory cache). In all cases, SoftPM was incorporated with minimal developer effort, was able to store and recover data successfully, and provide significant performance speedup (e.g., up to 10X for memcachedb and 83% for SQLite).

1 Introduction

Persistence of in-memory data is necessary for many classes of software including metadata persistence in systems software [21, 24, 32, 33, 35, 38, 40, 47, 48, 52, 59], application data persistence in in-memory databases and key-value stores [3, 5], and computational state persistence in high-performance computing (HPC) applications [19, 44]. Currently such software relies on the persistence primitives provided by the operating system (file or block I/O) or a database system. When using OS primitives, developers need to carefully track persistent data structures in their code and ensure the atomicity of persistent modifications. Additionally they are required to implement serialization/deserialization for their structures, potentially creating and managing additional metadata whose modifications must also be made consistent with the data they represent. On the other hand, using databases for persistent metadata is generally not an option within systems software, and when their use is possible, developers must deal with data impedance mismatch [34]. While some development complexity is alleviated by object-relation mapping libraries [10], these translators increase overhead along the data path. Most importantly, all of these solutions require substantial application involvement for making data persistent which ultimately increases code complexity affecting reliability, portability, and maintainability.

In this paper, we present *Software Persistent Memory* (SoftPM), a lightweight persistent memory abstraction for C. SoftPM provides a novel form of *orthogonal persistence* [8], whereby the persistence of data (the *how*) is seamless to the developer, while allowing effortless control over *when* and *what* data persists. To use SoftPM, developers create one or more persistent *containers* to house a subset of in-memory data that they wish to make persistent. They only need to ensure that a container's *root structure* houses pointers to the data structures they wish to make persistent (e.g. the head of a list or the root of a tree). SoftPM automatically discovers data reachable from a container's root structure (by recursively following pointers) and makes all new and modified data persistent. Restoring a container returns the container root structure from which all originally reachable data can be accessed. SoftPM thus obviates the need for explicitly managing persistent data and places no restrictions on persistent data locations in the process' address space. Finally, SoftPM improves I/O performance by eliminating the need to serialize data and by using a novel *chunk-remapping* technique which utilizes the property that all container data is memory resident and trades writing additional data for reducing overall I/O latency.

We evaluated a Linux prototype of SoftPM for correctness, ease of use, and performance using microbenchmarks and three real world applications including a recoverable distributed MPI application, SQLite [5] (a serverless database), and memcachedb (a distributed memory cache). In all cases, we could integrate SoftPM with little developer effort and store and recover application data successfully. In comparison to explicitly managing persistence within code, development complexity substantially reduced with SoftPM. Performance improvements were up to 10X for memcachedb and 83% for SQLite, when compared to their native, optimized, persistence implementations. Finally, for a HPC-class matrix multiplication application, SoftPM's asyn-

Function	Description
`int pCAlloc(int magic, int cSSize, void ** cStruct)`	*create a new container; returns a container identifier*
`int pCSetAttr(int cID, struct cattr * attr)`	*set container attributes; reports success or failure*
`struct cattr * pCGetAttr(int magic)`	*get attributes of an existing container; returns container attributes*
`void pPoint(int cID)`	*create a persistence point asynchronously*
`int pSync(int cID)`	*sync-commit outstanding persistence point I/Os; reports success or failure*
`int pCRestore(int magic, void ** cStruct)`	*restore a container; populates container struct, returns a container identifier*
`void pCFree(int cID)`	*free all in-memory container data*
`void pCDelete(int magic)`	*delete on-disk and in-memory container data*
`void pExclude(int cID, void * ptr)`	*do not follow pointer during container discovery*

Table 1: The SoftPM application programmer interface.

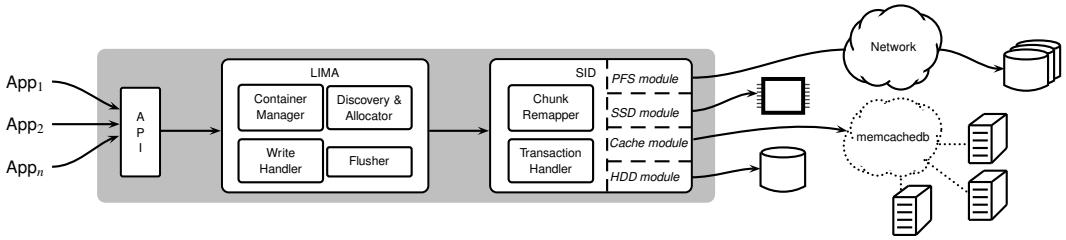


Figure 1: The SoftPM architecture.

Container Root Structure	Usage
`struct c_root { list_t *l; } *cr;`	`id = pCAlloc(m,sizeof(*cr),&cr); cr->l = list_head; pPoint(id);`

Figure 2: Implementing a persistent list. `pCAlloc` **allocates a container and** `pPoint` **makes it persistent.**

chronous persistence feature provided performance at close to memory speeds without compromising data consistency and recoverability.

2 SoftPM Overview

SoftPM implements a persistent memory abstraction called *container*. To use this abstraction, applications create one or more containers and associate a *root structure* with each. When the application requests a persistence point, SoftPM calculates a memory closure that contains all data reachable (recursively via pointers) from the container root, and writes it to storage atomically and (optionally) asynchronously.

The container root structure serves two purposes: *(i)* it frees developers from the burden of explicitly tracking persistent memory areas, and *(ii)* it provides a simple mechanism for accessing all persistent memory data after a restore operation. Table 1 summarizes the SoftPM API. In the simplest case, an application would create one container and create *persistence points* as necessary (Figure 2). Upon recovery, a pointer to a valid container root structure is returned.

2.1 System Architecture

The SoftPM API is implemented by two components: the *Location Independent Memory Allocator* (LIMA), and the *Storage-optimized I/O Driver* (SID) as depicted in Figure 1. LIMA's *container manager* handles container creation. LIMA manages the container's persistent data as a collection of memory pages *marked* for persistence. When creating a persistence point, the *discovery and allocator module* moves any data newly made reachable from the container root structure and located in volatile memory to these pages. Updates to these pages are tracked by the *write handler* at the granularity of multi-page *chunks*. When requested to do so, the *flusher* creates persistence points and sends the dirty chunks to the SID layer in an asynchronous manner. Restore requests are translated into chunks requests for SID.

The SID layer atomically commits container data to persistent storage and tunes I/O operations to the underlying storage mechanism. LIMA's flusher first notifies the *transaction handler* of a new persistence point and submits dirty chunks to SID. The *chunk remapper* implements a novel I/O technique which uses the property that all container data is memory resident and trades writing additional data for reducing overall I/O latency. We designed and evaluated SID implementations for hard drive, SSD, and memcached back-ends.

3 LIMA Design

Persistent containers build a foundation to provide seamless memory persistence. Container data is managed within a contiguous container virtual address space, a self-describing unit capable of being migrated across systems and applications running on the same hardware architecture. The container virtual address space is composed solely of pages marked for persistence including those containing application data and others used to store LIMA metadata. This virtual address space is mapped to

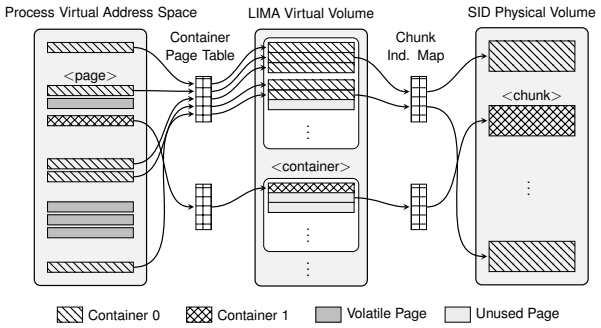


Figure 3: Container virtual address spaces in relation to process virtual address space and LIMA/SID volumes. The container virtual address space is chunked, containing a fixed number of pages (three in this case).

logically contiguous locations within the virtual volume managed by LIMA. SID remaps LIMA virtual (storage) volumes at the chunk granularity to the physical (storage) volume it manages. This organization is shown in Figure 3. The indirection mechanism implemented by SID simplifies persistent storage management for LIMA which can use a logically contiguous store for each container.

3.1 Container Manager

The container manager implements container allocation and restoration. To allocate a new container (`pCAlloc`), an in-memory container page table, that manages both application persistent data and LIMA metadata, is first initialized. Next, the container root structure and other internal LIMA metadata structures are initialized to be managed via the container page table.

To restore a container, an in-memory container instance is created and all container data and metadata loaded. Since container pages would likely be loaded into different portions of the process' address space, two classes of updates must be made to ensure consistency of the data. First, the container metadata must be updated to reflect the new in-memory data locations after the restore operation. Second, all memory pointers within data pages need to be updated to reflect the new memory addresses (pointer swizzling). To facilitate this, pointer locations are registered during process execution; we discuss automatic pointer detection in §5.

3.2 Discovery and Memory Allocation

A core feature of SoftPM is its ability to discover container data automatically. This allows substantial control over what data becomes persistent and frees the developer from the tedious and error-prone task of precisely specifying which portions of the address space must be allocated persistently. SoftPM implements automatic container discovery and persistent memory allocation by automatically detecting pointers in process memory, recursively moving data reachable from the container root to the container data pages, and fixing any *back references* (other pointers) to the data that was moved. In our implementation, this process is triggered each time a persistence point is requested by the application and is executed atomically by blocking all threads of a process only until the container discovery phase is completed; disk I/O is performed asynchronously (§3.4).

To make automatic container discovery possible, SoftPM uses static analysis and automatic source translation to register both *pointers* and *memory allocation* requests (detailed in §5). At runtime, pointers are added either to a *persistent pointer set* or a *volatile pointer set* as appropriate, and information about all memory allocations is gathered. Before creating a persistence point, if a pointer in the persistent pointer set (except those excluded using `pExclude`) references memory outside the container data pages, the allocation containing the address being referenced is moved to the persistent memory region. Forward pointers contained within the moved data are recursively followed to similarly move other new reachable data using an edge-marking approach [30]. Finally, back references to all the data moved are updated. This process is shown in Figure 4. There are two special cases for when the target is not within a recognized allocation region. If it points to the code segment (e.g. function pointers), the memory mapped code is registered so that we can "fix" the pointer on restoration. Otherwise, the pointer metadata is marked so that its value is set to NULL when the container gets restored; this allows SoftPM to correctly handle pointers to OS state dependent objects such as files and sockets within standard libraries. If allocations made by library code are required to be persistent, then the libraries must also be statically translated using SoftPM; the programmer is provided with circumstantial information to help with this. In many cases, simply reinitializing the library upon restoration is sufficient, for instance, we added one extra line in SQLite (see § 6.3.3) for library re-initialization.

3.3 Write Handler

To minimize disk I/O, SoftPM commits only modified data during a persistence point. The *write handler* is responsible for tracking such changes. First, sets of contiguous pages in the container virtual address space are grouped into fixed-size *chunks*. At the beginning of a persistence point, all container data and metadata pages are marked *read-only*. If any of these pages are subsequently written into, two alternatives arise when handing the fault: *(i)* there is no persistence point being created currently – in this case, we allow the write, mark the chunk *dirty*, and its pages *read-write*. This ensures at most one write page fault per chunk between two consec-

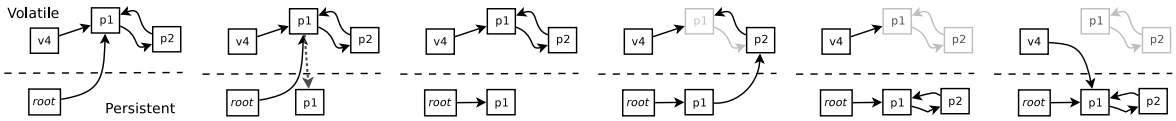


Figure 4: Container Discovery. Grey boxes indicate freed memory.

utive persistence points. *(ii)* there is a persistence point being created currently – then we check if the chunk has already been made persistent. If so, we simply proceed as in the first case. If it has not yet been made persistent, a copy of the chunk is first created to be written out as part of the ongoing persistence point, while write to the original chunk is handled as in the first case.

3.4 Flusher

Persistence points are created asynchronously (via `pPoint`) as follows. First, the flusher waits for previous persistence points for the same container to finish. It then temporarily suspends other threads of the process (if any) and marks all the pages of the container as read-only. If no chunks were modified since the previous persistence point, then no further action is taken. If modifications exist, the flusher spawns a new thread to handle the writing, sets the state of the container to *persistence point commit*, and returns to the caller after unblocking all threads. The handler thread first identifies all the dirty chunks within the container and issues write operations to SID. Once all the chunks are committed to the persistent store, SID notifies the flusher. The flusher then reverts the state of the container to indicate that persistence point has been committed.

4 SID Design

LIMA maps chunks and containers to its logical volume statically and writes out only the modified chunks during persistence points. If a mechanical disk drive is used directly to store this logical volume, I/O operations during a persistence point can result in large seek and rotational delay overheads due to fragmented chunk writes within a single container; if multiple containers are in use simultaneously, the problem compounds causing disk head movement across multiple container boundaries. If a solid-state drive (SSD) were used as the persistent store, the LIMA volume layout will result in undesirable random writes to the SSD that is detrimental to both I/O performance and wear-leveling [22, 31]. The complementary requirement of ensuring atomicity of all chunk writes during a persistence point must be addressed as well. The SID component of SoftPM is an indirection layer below LIMA and addresses the above concerns.

4.1 SID Basics

SID divides the physical volume into chunk-sized units and maps chunks in the LIMA logical volume to physical volume locations for I/O optimization. The *chunk remapper* utilizes the property that all container data is memory resident and trades writing additional data (chunk granularity writes) for reducing I/O latency using device-specific optimizations.

Each physical volume stores *volume-level SID metadata* at a fixed location. This metadata includes for each container the address of a single physical chunk which stores two of the most recent versions of metadata for the container to aid crash recovery (elaborated later). To support chunk indirection, SID maintains a *chunk indirection map* as part of the container metadata. Finally, SID also maintains both an in-memory and on-disk per-container *free chunk bitmap* to locate the chunks utilized by a container. We chose to store per-container free chunk bitmaps to make each container self-describing and as a simple measure to eliminate race conditions when persisting multiple containers simultaneously.

During SID initialization, the free chunk bitmaps for each container stored on the physical volume are read into memory. An in-memory *global* free chunk bitmap obtained by merging the per-container free chunk bitmaps is used to locate free chunks in the physical volume quickly during runtime.

Atomic Persistence. To ensure atomicity of all chunk writes within a persistence point, SID uses persistence *version* numbers. When SID receives a request to create a persistence point, it goes through several steps in sequence. First, it writes all the dirty data chunks; chunks are never updated in place to allow recovery of the previous version of the chunks in case the persistence operation cannot be completed. Once the data chunk writes have all been acknowledged, SID writes the updated free chunk bitmap. Finally, it writes the container's metadata. This metadata includes, the chunk indirection map, the location of the newly written free chunk bitmap, and a (monotonically increasing) version number to uniquely identify the persistence point. Writing the last block of the metadata (the version number) after an I/O barrier commits the persistence point to storage; we reasonably assume that this block gets written to the storage device atomically.

Recovery. SID recovers the same way after both normal shutdowns and crashes. In either case, it identifies the most recent metadata for each container by inspecting their version numbers. It then reads the per-container free chunk bitmaps, and builds the global free chunk bitmap by merging all per-container bitmaps. When the application requests to restore a container, the most recent version of the chunk indirection map is used to reconstruct the container data in memory.

4.2 Device-specific optimizations

Disk Drives. Since sequential access to disk drives is orders of magnitude more efficient than random, we designed a mechanical disk SID driver to employ mostly-sequential chunk layout. The design assumes that the storage device will be performance rather than capacity bound, justifying a fair degree of space over-provisioning for the SID physical volume. Every chunk is written to the nearest free location succeeding the previously written location, wrapping around in a circular fashion. The greater the over-provisioning of the SID physical volume, the higher the probability of finding an adjacent free chunk. For instance, a 1.5X over-provisioning of capacity will result in every third chunk being free on average. Given sufficient outstanding chunk requests in the disk queue at any time, chunks can be written with virtually no seek overhead and minimum rotational delay. Reclaiming free space is vastly simpler than a log-structured design [49] or that of other copy-on-write systems like WAFL [28] because *(i)* the design is not strictly log-structured and does not require multiple chunk writes to be sequential, and *(ii)* reclaiming obsolete chunks is as simple as updating a single bit in the freespace bitmap without the need for log cleaning or garbage collection that can affect performance.

Flash drives. An SSD's logical address space is organized into *erase units* which were hundreds of kilobytes to a few megabytes in size for the SSD units we tested. If entire erase units are written sequentially, free space can be garbage collected using inexpensive *switch merge* operations rather than more expensive *full merge* operations that require data copying [31]. SID writes to the SSD space one erase unit at a time by tuning its chunk size to a multiple of the erase unit size. The trade-off between the availability of free chunks and additional capacity provisioning follows the same arguments as those for disk drives above.

5 Pointer Detection

As discussed in §3, LIMA must track pointers in memory for automatic container discovery and updating pointer values during container restoration. The life-cycle of a pointer can be defined using the following stages: *(i) allocation*: when memory to store the pointer is allocated, *(ii) initialization*: when the value of the pointer is initialized, *(iii) use*: when the pointer value is read or written, and *(iv) deallocation*: when the memory used to store the pointer is freed. Note that, a pointer is always associated with an allocation. In SoftPM, we detect pointers at *initialization*, both explicitly (via assignment) or implicitly (via memory copying or reallocation). Hence, if programs make use of user-defined memory management mechanisms (e.g., allocation, deallocation, and copy), these must be registered with SoftPM to be correctly accounted for.

SoftPM's pointer detection works in two phases. At compile time, a static analyzer based on CIL (C Intermediate Language) [43] parses the program's code looking for instructions that allocate memory or initialize pointers. When such instructions are found, the analyzer inserts static hints so that these operations are registered by the SoftPM runtime. At runtime, SoftPM maintains an *allocation table* with one entry per active memory allocation. Each entry contains the address of the allocation in the process' address-space, size, and a list of pointers within the allocation. Pointers are added to this list upon initialization which can be done either *explicitly* or *implicitly*. A pointer can be initialized explicitly when it appears as an *l-value* of an assignment statement. Second, during memory copying or moving, any initialized pointers present in the source address range are also considered as implicitly initialized in the destination address range. Additionally, the source allocation and pointers are deregistered on memory moves. When memory gets deallocated, the entry is deleted from the allocation table and its pointers deregistered.

Notes. Since SoftPM relies on static type information to detect pointers, it cannot record integers that may be (cast and) used as pointers by itself. However, developers can insert static hints to the SoftPM runtime about the presence of additional "intentionally mistyped" pointers to handle such oddities. Additionally, SoftPM is agnostic to the application's semantics and it is not intended to detect arbitrary memory errors. However, SoftPM itself is immune to most invalid states. For example, SoftPM checks whether a pointer's target is a valid region as per the memory allocation table before following it when computing the memory closure during container discovery. This safeguard avoids bloating the memory closure due to "rogue" pointers. We discuss this further detail in § 8.

Related work. Pointer detection is an integral part of garbage collectors [58]. However, for languages that are not strongly typed, conservative pointer detection is used [12]. This approach is unsuitable for SoftPM since it is necessary to swizzle pointers. To the best of our knowledge, the static-dynamic hybrid approach to *exact*

pointer detection presented in this paper, is the first of its kind. Finally, although pointer detection seems similar to *points-to* analysis [27], these are quite different in scope. The former is concerned about if a given memory *location* contains a valid memory address, while the latter is concerned about *exactly which* memory addresses a memory location can contain.

6 Evaluation

Our evaluation seeks to address the correctness, ease of use, and performance implications of using SoftPM. We compare SoftPM to conventional solutions for persistence using a variety of different application benchmarks and microbenchmarks. In cases where the application had a built-in persistence routine, we compared SoftPM against it using the application's default configuration. Where such an implementation was not available, we used the TPL serialization library [6] v1.5 to implement the serialization of data structures. All experiments were done on one or more 4-Core AMD Opteron 1381 with 8 GB of RAM using WDC WD5002ABYS SATA and MTRON 64GB SSD drives running Linux 2.6.31.

6.1 Workloads

We discuss workloads that are used in the rest of this evaluation and how we validated the consistency of persistent containers stored using SoftPM in each case.

Data Structures. For our initial set of experiments we used the DragonFly BSD [1] v2.13.0 implementation of commonly used data structures including arrays, lists, trees, and hashtables. We populated these with large number of entries, queried, and modified them, creating persistence points after each operation.

Memcachedb [3]. A persistent distributed cache based on memcached [2] which uses Berkeley DB (BDB) [45] v4.7.25 to persistently store elements of the cache. Memcachedb v1.2.0 stores its key-value pairs in a BDB database, which provides a native persistent key value store by using either a btree or a hash table. We modified memcachedb to use a hash table which we make persistent using SoftPM instead of using BDB, and compared its performance to the native version using default configurations of the software. To use SoftPM with memcachedb, we modified the file which interfaced with BDB, reducing the LOC from 205 to 40. The workload consisted of inserting a large number of key-value pairs into memcachedb and performing a number of lookups, inserts, and deletes of random entries, creating persistence points after each operation.

SQLite [5]. A popular serverless database system with more than 70K LOC. We modified it to use SoftPM for persistence and compared it against its own persistence routines. SQLite v3.7.5 uses a variety of complex data structures to optimize inserts and queries among other operations; it also implements and uses a custom slab-based memory allocator. A simple examination of the SQLite API revealed that all the database metadata and data is handled through one top-level data structure, called `db`. Thus, we created a container with just this structure and excluded an incorrectly detected pointer resulting from casting an `int` as a `void*`. In total, we added 9 LOC to make the database persistent using SoftPM which include a few more code to re-initialize a library.

MPI Matrix Multiplication. A recoverable parallel matrix multiplication that uses Open MPI v1.3.2 and checkpoints state across processes running on multiple machines.

6.2 Correctness Evaluation

To evaluate the correctness of SoftPM for each of the above applications, we crashed processes at random execution points and verified the integrity of the data when loaded from the SoftPM containers. We then compared what was restored from SoftPM to what was loaded from the native persistence method (e.g. BDB or file); in all cases, the contents were found to be equal. Finally, given that we were able to examine and correctly analyze complex applications such as SQLite with a large number of dynamically allocated structures, pointers, and a custom memory allocation implementation, we are confident that our static and dynamic analysis for pointer detection is sound.

6.3 Case Studies

In this section, we perform several case studies including *(i)* a set of SoftPM-based persistent data structures, *(ii)* an alternate implementation of memcachedb [3] which uses SoftPM for persistence, *(iii)* a persistent version of SQLite [5], a serverless database based on SoftPM, and *(iv)* a recoverable parallel matrix multiplication application that uses MPI.

6.3.1 Making Data Structures Persistent

We examined several systems that require persistence of in-memory data and realized that these systems largely used well-known data structures to store their persistent data such as arrays, lists, trees, and hashtables. A summary of this information is presented in Table 2. We constructed several microbenchmarks that create and modify several types of data structures using SoftPM and TPL [6], a data structure serialization library. To quantify the reduction in development complexity we compared the lines of code necessary to implement persistence for various data structures using both solutions. We report in Table 3 the lines of code (LOC) without any persistence and the additional LOC when implementing persistence using TPL and SoftPM respectively.

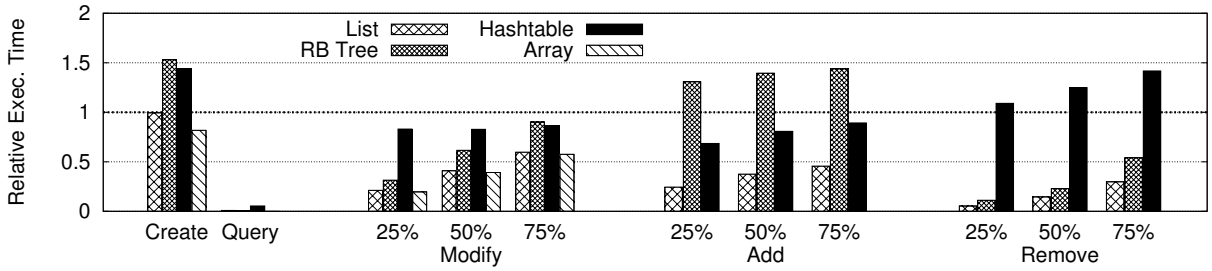


Figure 5: Performance of individual data structure operations. The bars represent the execution time of the SoftPM version relative to a version that uses the TPL serialization library for persistence. We used fixed size arrays which do not support add or remove operations.

Systems	Arrays	Lists	Hash Tables	Trees	C
BORG [11]	✓		✓	✓	✓
CDP [33]				✓	
Clotho [21]	✓	✓			
EXCES [54]	✓	✓	✓	✓	
Deduplication [59]	✓		✓		
FlaZ [38]	✓	✓	✓		
Foundation [48]	✓		✓		
GPAW [41]	✓				
I/O Shepherd [24]	✓			✓	
I/O Dedup [32]	✓			✓	✓
Venti [47]			✓		

Table 2: Persistent structures used in application and systems software. Arrays are multidimensional in some cases. *C* indicates other complex (graphs and/or hybrid) structures were used. This summary is created based on descriptions within respective articles and/or direct communication with the developers of these systems.

For each data structure we perform several operations (e.g modify) and make the data structure persistent. Note that the TPL version writes entire structures to disk, whereas SoftPM writes only what was modified. For *create*, SoftPM calculates the memory closure, move the discovered data to persistent memory, and write to disk and overhead is proportional to this work. The *query* operation doesn't modify any data and SoftPM clearly outperforms TPL in this case. *modify* only changes existing data values, *remove* reduces the amount of data written by TPL and involves only metadata updates in SoftPM, and *add* increases the size of the data structure increasing both the amount of data and metadata writes with SoftPM. Figure 5 presents the execution times of the SoftPM version relative to the TPL version. Two interesting points are evidenced here. First, for add operations SoftPM outperforms TPL for all data structures except RB Tree, this is due to balancing of the tree modifying almost the entire data structure in the process requiring expensive re-discovery, data movement, and writing. Second, the remove operations for Hashtable are expensive for SoftPM since its implementation uses the largest number of pointer; removing involves a linear search in one of our internal data structures and we are currently working on optimizing this.

Data Structure	Original LOC	LOC for Persistence	LOC to use SoftPM
Array	102	17	3
Linked List	188	24	3
RB Tree	285	22	3
Hash Table	396	21	3
SQLite	73042	6696	9
memcachedb	1894	205	40

Table 3: Lines of code to make structures (or applications) persistent and recover them from disk. We used TPL for Array, Linked List, RB Tree, and Hash Table; SQLite and memcachedb implement custom persistence.

6.3.2 Comparing with Berkeley DB

memcachedb is an implementation of memcached which periodically makes the key value store persistent by writing to a Berkeley DB (BDB) [45] database. BDB provides a persistent key value store using a btree (BDB-Btree) or hash table (BDB-HT), as well as incremental persistence by writing only dirty objects, either synchronously or asynchronously. We modified memcachedb to use a hash table which we make persistent using SoftPM instead of using BDB. In Figure 6 we compare the operations per second achieved while changing the persistence back-end. SoftPM outperforms both variants of BDB by upto 2.5X for the asynchronous versions and by 10X for the synchronous.

6.3.3 Making an in Memory Database Persistent

SQLite is a production-quality highly optimized serverless database, it is embedded within many popular software such as Firefox, iOS, Solaris, and PHP. We implemented a benchmark which creates a database and performs random insert, select, update, and delete transactions. We compare the native SQLite persistence to that using SoftPM; transactions are synchronous in both cases. Figure 7 shows that SoftPM is able to achieve 55%

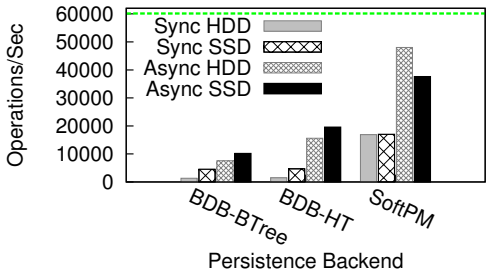


Figure 6: Performance of memcachedb using different persistent back-ends. The workload randomly adds, queries, and deletes 512 byte elements with 16 byte keys. The dashed line represents a memory only solution.

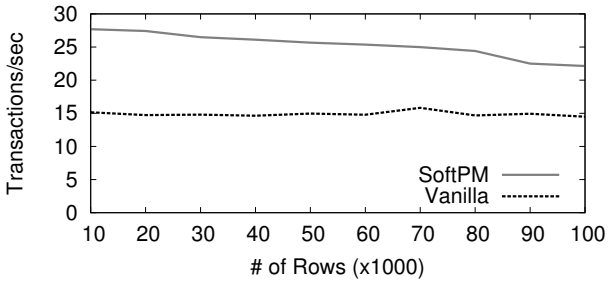


Figure 7: SQLite transactions per second comparison when using SoftPM and the native file persistence.

to 83% higher transactions rate depending on the size of the database. We believe this is a significant achievement for SoftPM given two facts. First, SQLite is a large and complex code base which includes a complete stand alone database application and second, SQLite's file transactions are heavily optimized and account for more than 6K LOC. Further analysis revealed that most of SoftPM's savings arise from its ability to optimize I/O operations relative to SQLite. The reduction in performance improvement with a larger number of rows in the database is largely attributable to a sub-optimal container discovery implementation; by implementing incremental discovery to include only those pointers within dirty pages, we expect to scale performance better with database size in future versions of SoftPM. Figure 8 shows a breakdown of the total overhead including I/O time incurred by SoftPM which are smaller than the time taken by the native version of SQLite. Finally, all of this improvement was obtained with only 9 additional LOC within SQLite to use SoftPM, a significant reduction relative to its native persistence implementation (6696 LOC).

6.3.4 Recoverable Parallel Matrix Multiplication

To compare SoftPM's performance to conventional checkpointing methods, we implemented a parallel matrix multiplication application using Cannon's algorithm [25]. We evaluated multiple solutions, including a no checkpoint non-recoverable implementation, a serialization-based implementation which serializes the matrices to files, and *sync* and *async* versions of SoftPM, in all cases a checkpoint is made after calculating each sub-matrix. For the file-based checkpointing version we added 79 LOC to serialize, write the matrix to a file, and recover from the file. In the SoftPM version, we added 44 LOC, half of them for synchronization across processes to make sure all processes restored the same version after a crash.

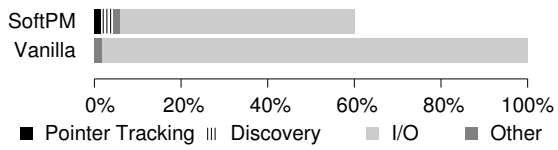


Figure 8: Breakdown of time spent in the SQLite benchmark for 100K rows.

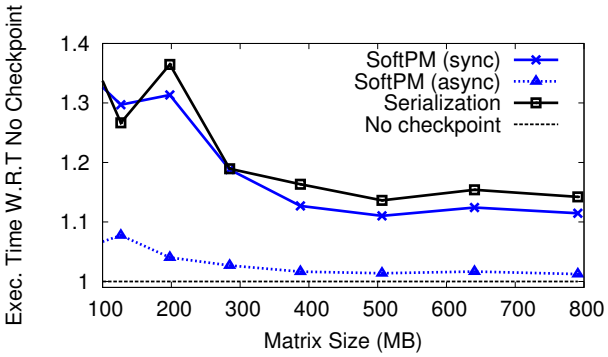


Figure 9: Contrasting application execution times for MPI matrix multiplication using 9 processes.

Figure 9 compares the total execution time across these solutions. Synchronous SoftPM and the serialization solution have similar performance. Interestingly, because of unique ability of overlapping checkpoints with computation, the asynchronous version of SoftPM performs significantly better than either of the above, in fact, within a 1% difference (for large matrices) relative to the memory-only solution.

6.4 Microbenchmarks

In this section, we evaluate the sensitivity of SoftPM performance to its configuration parameters using a series of microbenchmarks. For these experiments, we used a persistent linked list as the in-memory data structure. Where discussed, *SoftPM* represents a version which uses a SoftPM container for persistence; TPL represents an alternate implementation using the TPL serialization library. Each result is averaged over 10 runs, and except when studying its impact, the size of a chunk in the SID

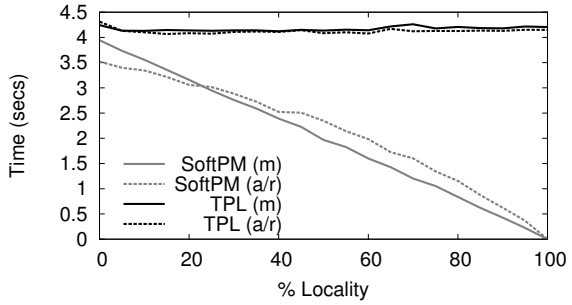


Figure 10: Impact of locality on incremental persistence. Two different sets of experiments are performed: *(m)* where only the contents of the nodes are modified, and *(a/r)* where nodes are added and removed from the list. In both cases the size of the list is always 500MB.

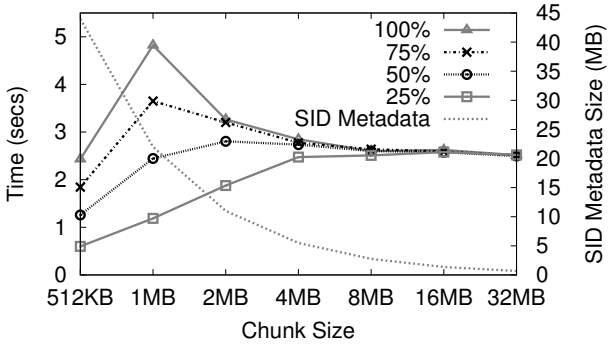


Figure 11: Impact of chunk size on persistence point time. A list of size 500MB is made persistent and individual lines depict for a specific fraction of the list modified.

layer is set to 512KB. To make the linked list persistent, SoftPM and TPL add 5 and 28 LOC, respectively.

Incremental Persistence. Usually, applications modify only a subset of the in-memory data between persistence points. SoftPM implements incremental persistence by writing only the modified chunks, which we evaluated by varying the locality of updates to a persistent linked list, shown in Figure 10. As expected, TPL requires approximately the same amount of time regardless of how much data is modified; it always writes the entire data structure. The SoftPM version requires less time to create persistence points as update locality increases.

Chunk Size. SoftPM tracks changes and writes container data at the granularity of a chunk to create persistence points. When locality is high, smaller chunks lead to lesser data written but greater SID metadata overhead because of a bigger chunk indirection map and free chunk bitmap. On the other hand, larger chunks imply more data written but less SID metadata. Figure 11 shows the time taken to create persistence points and the size of the SID metadata at different chunk sizes.

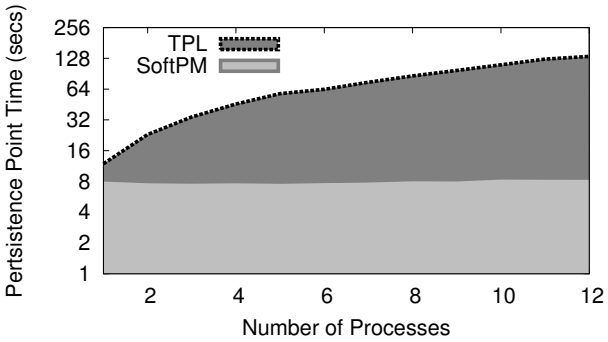


Figure 12: Time to create persistence points for multiple parallel processes. Every process persists a list of size (1GB/number-of-processes).

Parallel Persistence Points. The SID layer optimizes the way writes are performed to the underlying store, e.g. writing to disk drives semi-sequentially. Figure 12 depicts the performance of SoftPM in relation to TPL when multiple processes create persistence points to different containers at the same time. We vary the number of processes, but keep the total amount of data persisted by all the processes a constant. The total time to persist using SoftPM is a constant given that the same amount of data is written. On the other hand, the time for TPL increases with the number of threads, because of lack of optimization of the interleaving writes to the different container files at the storage level.

Percentage of Pointers in Data. Creating a persistence point requires computing a transitive memory closure, an operation whose time complexity is a function of the number of pointers in container data. We varied the fraction of the memory (used by the linked list) that is used to store pointers (quantified as "percentage pointers in data") and measured the time to create a full (non-incremental) persistence point.

We compare performance with a TPL version of the benchmark that writes only the contents of the elements of the list to a file in sequence without having to store pointers. A linked list of total size 500MB was used. Figure 13 shows the persistence point creation times when varying the percentage pointers in data. SoftPM is not always more efficient in creating persistence points than TPL, due to the need to track and store all the pointers and the additional pointer data and SoftPM metadata that needs to be written to storage. The linked list represents one of the best case scenarios for the TPL version since the serialization of an entire linked list is very simple and performs very well due to sequential writing. We also point out here that we are measuring times for registering pointers in the entire list, a full discovery and (non-incremental) persistence, a likely worst case for SoftPM; in practice, SoftPM will track pointers incrementally and

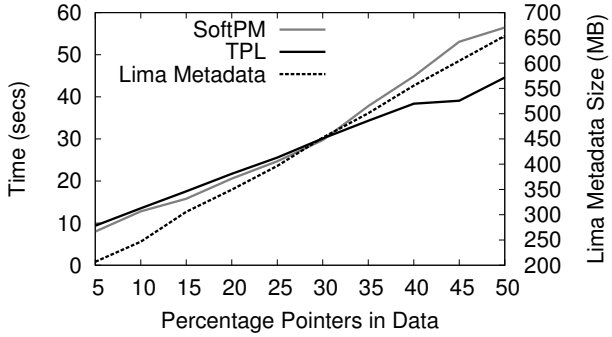


Figure 13: Time to persist a linked list and LIMA metadata size, varying percentage of pointers in data. The total size is fixed at 500MB and node sizes are varied accordingly.

persist incrementally as the list gets modified over time. Further, for the complex programs we studied the percentage pointers in data is significantly lower; in SQLite this ratio was 4.44% and for an MPI-based matrix multiplication this ratio was less than 0.04%. Finally, the amount of SoftPM metadata per pointer can be further optimized; instead of 64 bit pointer locations (as we currently do), we can store a single page address and multiple 16 bit offsets.

7 Related Work

Persistence techniques can be classified into *system-managed*, *application-managed*, and *application-directed*. System-managed persistence is usually handled by a library with optional OS support. In some solutions, it involves writing a process's entire execution state to persistent storage [23, 26, 13]. Other solutions implement persistently mapped memories for programs with pointer swizzling at page fault time [51]. While transparent to developers, this approach lacks the flexibility of separating persistent and non-persistent data required by many applications and systems software. With application-managed persistence [19, 44], application developers identify and track changes to persistent data and build serialization-based persistence and restoration routines. Some hybrid techniques implemented either as persistent memory libraries and persistent object stores have combined reliance on extensive developer input about persistent data with system-managed persistence [14, 20, 36, 34, 46, 50]. However, these solutions involve substantial development complexity, are prone to developer error, and in some cases demand extensive tuning of persistence implementations to the storage system making them less portable. For instance, ObjectStore[34] requires developers to specify which allocations are persistent and their type by overloading the new operator in C++ [4].

Application-directed persistence provides a middle ground. The application chooses what data needs to be persistent, but a library implements the persistence. The earliest instances were persistent object storage systems [16] based on Atkinson's seminal *orthogonal persistence* proposal [8]. Applications create objects and explicit inter-object references, and the object storage system (de)serializes entire objects and (un)swizzles reference pointers [42]. Some persistent object systems (e.g., Versant Persistent Objects [7], SSDAlloc [9], Dali [29]) eliminate object serialization but they require (varying degrees of) careful development that includes identifying and explicitly tagging persistent objects, identifying and (un)swizzling pointers, converting strings and arrays in the code to custom persistent counterpart types, and tagging functions that modify persistent objects.

Recoverable Virtual Memory (RVM) [50] was one of the first to demonstrate the potential for memory-like interfaces to storage. However, its approach has some key limitations when compared to SoftPM. First, RVM's interface still requires substantial developer involvement. Developers must track all persistent data, allocate these within RVM's persistent region, and ensure that dependence relations among persistent data are satisfied (e.g., if persistent structure a points to b, then b must also be made persistent). Manually tracking such relations is tedious and error-prone. Further, developers must specify the address ranges to be modified ahead of time to optimize performance. These requirements were reported to be the source of most programmer bugs when using RVM [39]. Second, RVM's static mapping of persistent memory segments makes it too rigid for contemporary systems that demand flexibility in managing address spaces [37, 53]. In particular, this approach is not encouraged in today's commodity operating systems that employ address-space layout randomization for security [53]. Finally, RVM is also restrictive in dynamically growing and shrinking persistent segments and limits the portability of a persistent segment due to its address range restrictions.

The recent Mnemosyne [56] and NV-Heaps [15] projects also provide persistent memory abstractions similar to SoftPM. However, there are at least two key differences. First, both of the solutions are explicitly designed for non-volatile memories or NVM (e.g., phase-change memory) that are not yet commercially available. Most significantly, these devices are intended to be CPU accessible and byte addressable which eliminates copying data in/out of DRAM [17]. Thus, the focus of these systems is on providing consistent updates to NVM-resident persistent memory via transactions. On the other hand, SoftPM targets currently available commodity technology. Second, neither of these

systems provide the *orthogonal persistence* that SoftPM enables; rather, they require the developer to explicitly identify individual allocations as persistent or not and track and manage changes to these within transactions. For instance, the NV-Heaps work argues that explicit tracking and notification of persistent data ensures that the developer does not inadvertently include more data than she intends [15]. We take the converse position that besides making persistence vastly simpler to use, automatic discovery ensures that the developer will not inadvertently exclude data that does need to be persistent for correctness of recovery, while simultaneously retaining the ability to explicitly exclude portions of data when unnecessary. Further, SoftPM's design, which relies on interposing on application memory allocations, ensures that pointers to library structures (e.g., files or sockets) are reset to NULL upon container restoration by default, thus relieving the developer of explicitly excluding such OS dependent data; such OS specific data is typically re-initialized upon application restart. Finally, feedback about automatically discovered persistent containers from SoftPM can help the developer in reasoning about and eliminating inadvertently included data.

Single level persistent stores as used in the Grasshopper operating system [18] employ pointer swizzling to convert persistent store references to in-memory addresses at the page granularity [55, 57] by consulting an object table within the object store or OS. Updates to persistent pointers are batch-updated (swizzled) when writing pages out. SoftPM fixes pointer addresses when persistent containers get loaded into memory but is free of swizzling during container writing time.

Finally, Java objects can be serialized and saved to persistent storage, from where it can be later loaded and recreated. Further, the Java runtime uses its access to the object's specification, unavailable in other lower-level imperative languages that SoftPM targets.

7.1 SoftPM: A New Approach

SoftPM implements application-directed persistence and differs from the above body of work in providing a solution that: requires little developer effort, works with currently available commodity storage, is flexible enough to apply to modern systems, and enables memory to be ported easily across different address space configurations and applications. Unlike previous solutions in the literature, SoftPM automatically discovers all the persistent data starting from a simple user-defined root structure to implement orthogonal persistence. SoftPM's modular design explicitly optimizes I/O using chunk remapping and tuning I/Os for specific storage devices. Further, SoftPM's asynchronous persistence allows overlapping computation with persistence I/O operations. Finally, unlike most previous solutions, SoftPM implements persistence for the weakly typed C language, typically used for developing systems code using a novel approach that combines both static and dynamic analysis techniques.

8 Discussion and Future Work

Several issues related to the assumptions, scope, and current limitations of SoftPM warrant further discussion and also give us direction for future work.

Programmer errors. SoftPM's automatic discovery of updated container data depends on the programmer having correctly defined pointers to the data. One concern might be that if the programmer incorrectly assigned a pointer value, that could result in corrupt data propagating to disk or losing portions of the container. This is a form of programmer error to which SoftPM seems more susceptible to. However, such programmer errors would also affect other classes of persistence solutions including those based on data structure serialization since these also require navigating hierarchies of structures. Nevertheless, SoftPM does provide a straightforward resolution when such errors get exercised. While not discussed in this paper, the version of SoftPM that was evaluated in this paper implements container versioning whereby previously committed un-corrupted versions of containers can be recovered when such errors are detected. Additionally, we are currently implementing simple checks to warn the developer of unexpected states which could be indicators of such errors; e.g., a persistent pointer points to a non-heap location.

Container sharing. Sharing container data across threads within a single address-space is supported in SoftPM. Threads sharing the container would have to synchronize updates as necessary using conventional locking mechanisms. Sharing memory data across containers within a single address-space is also supported in SoftPM. These containers can be independently checkpointed and each container would store a persistent copy of its data. However, sharing container data persistently is not supported. Further, in our current implementation, containers cannot be simultaneously shared across process address-spaces. In the future, such sharing can be facilitated by implementing the SoftPM interface as library system calls so that container operations can be centrally managed.

Non-trivial types. SoftPM currently does not handle pointers that are either untyped or ambiguously typed. This can occur if a programmer uses a *special* integer type to store a pointer value or if a pointer type is part of a union. These can be resolved in the future with additional hints to SoftPM's static translator from the programmer. Additionally, the runtime could hint to SoftPM

about when a union type resolves to a pointer and when it is no longer so.

Unstructured data. The utility of SoftPM in simplifying development depends on the type of the data that must be made persistent. Unstructured data (e.g., audio or video streams) are largely byte streams and do not stand to benefit as much from SoftPM as data that has structure containing a number of distinct elements and pointers between them. Specifically, unstructured data tends not to get modified in place as much as structured data and consequently they may not benefit from the incremental change tracking that SoftPM implements.

9 Conclusion

For applications and systems that rely on a portion of their state being persistent to ensure correctness for continued operation, the availability of a lightweight and simple solution for memory persistence is valuable. SoftPM addresses this need by providing a solution that is both simple and effective. Developers use the existing memory interfaces as-is, needing only to instantiate persistent containers and container root structures besides requesting persistence points. They thus entirely bypass the complex requirements of identifying all persistent data in code, tracking modifications to them, and writing serialization and optimized persistence routines specific to a storage system. SoftPM automates persistence by automatically discovering data that must be made persistent for correct recovery and ensures the atomic persistence of all modifications to the container; storage I/O optimizations are modularized within SoftPM making it conveniently portable. Recovery of persistent memory is equally simple; SoftPM returns a pointer to the container root via which the entire container can be accessed. We evaluated SoftPM using a range of microbenchmarks, an MPI application, SQLite database, and a distributed memcachedb application. Development complexity as measured using lines of code was substantially reduced when using SoftPM relative to custom-built persistence of the application itself as well as persistence using an off-the-shelf serialization library. Performance results were also very encouraging with improvements of up to 10X, with SoftPM's asynchronous persistence feature demonstrating the potential for performing at close to memory speeds.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Paul Barham, whose feedback substantially improved our work. We are also grateful to Michail Flouris, Haryadi Gunawi, and Guy Laden for sharing the details of the persistent data structures they used in their systems. This work was supported by NSF grants CNS-0747038 and CCF-093796. Jorge Guerra was supported in part by an IBM PhD Fellowship.

References

[1] DragonFlyBSD. http://www.dragonflybsd.org/.

[2] memcached. http://memcached.org/.

[3] memcachedb. http://memcachedb.org/.

[4] ObjectStore Release 7.3 Documentation. http://documentation.progress.com/output/ostore/7.3.0/.

[5] SQLite. http://www.sqlite.org/.

[6] tpl. http://tpl.sourceforge.net/.

[7] Versant. http://www.versant.com/.

[8] M. P. Atkinson. Programming Languages and Databases. In *VLDB*, 1978.

[9] A. Badam and V. S. Pai. Ssdalloc: Hybrid ssd/ram memory management made easy. In *Proc. of NSDR*, 2009.

[10] D. Barry and T. Stanienda. Solving the java object storage problem. *Computer*, 31(11):33–40, 1998.

[11] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis. BORG: Block-reORGanization and Self-optimization in Storage Systems. In *Proc. of USENIX FAST*, 2009.

[12] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience* , 18(9):807–921, September 1988.

[13] G. Bronevetsky, D. Marques, K. Pingali, P. Szwed, and M. Schulz. Application-level Checkpointing for Shared Memory Programs. *SIGARCH Comput. Archit. News*, 32(5):235–247, 2004.

[14] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. In *Proceedings of ACM SIGMOD*, 1994.

[15] J. Coburn, A. Caulfield, A. Akel, L. Grupp, R. Gupta, R. Jhala, and S. Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proc. of ASPLOS*, 2011.

[16] W. P. Cockshott, M. P. Atkinson, K. J. Chisholm, P. J. Bailey, and R. Morrison. POMS - A Persistent Object Management System. *Software Practice and Experience*, 14(1):49– 71, 1984.

[17] 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 *Proc. of SOSP*, 2009.

[18] A. Dearle, R. di Bona, J. Farrow, F. Henskens, A. Lindström, J. Rosenberg, and F. Vaughan. Grasshopper: An Orthogonally Persistent Operating System. *Computer Systems*, 7(3):289–312, 1994.

[19] E. N. Elnozahy and J. S. Plank. Checkpointing for Peta-Scale Systems: A Look into the Future of Practical Rollback-Recovery. *IEEE TDSC*, 1(2):97–108, 2004.

[20] J. L. Eppinger. *Virtual Memory Management for Transaction Processing Systems*. PhD thesis, Carnegie Mellon University, 1989.

[21] M. D. Flouris and A. Bilas. Clotho: Transparent Data Ver-

sioning at the Block I/O Level. In *Proc. of IEEE MSST*, 2004.

[22] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37(2):138–163, 2005.

[23] R. Gioiosa, J. C. Sancho, S. Jiang, F. Petrini, and K. Davis. Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers. In *Proc. of the ACM/IEEE SC*, 2005.

[24] H. S. Gunawi, V. Prabhakaran, S. Krishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving File System Reliability with I/O Shepherding. In *Proc. of ACM SOSP*, 2007.

[25] H. Gupta and P. Sadayappan. Communication efficient matrix-multiplication on hypercubes. *Proc. of the ACM SPAA*, 1994.

[26] P. H. Hargrove and J. C. Duell. Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. In *Proc.of SciDAC Conference*, 2006.

[27] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE'01*, pages 54–61. ACM Press, 2001.

[28] D. Hitz, J. Lau, and M. Malcolm. File system design for an nfs file server appliance. In *Proc. of the USENIX Technical Conference*, 1994.

[29] H. V. Jagadish, D. F. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan. Dali: A high performance main memory storage manager. In *Proc. of VLDB*, 1994.

[30] S. V. Kakkad and P. R. Wilson. Address translation strategies in the texas persistent store. In *Proceedings of theUSENIX Conference on Object-Oriented Technologies & Systems*, 1999.

[31] A. Kawaguchi, S. Nishioka, and H. Motoda. A Flash-memory based File System. In *USENIX Technical*, 1995.

[32] R. Koller and R. Rangaswami. I/O deduplication: Utilizing content similarity to improve i/o performance. In *Proc. of USENIX FAST*, 2010.

[33] G. Laden, P. Ta-Shma, E. Yaffe, M. Factor, and S. Fienblit. Architectures for controller based cdp. In *Proc. of USENIX FAST*, 2007.

[34] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The objectstore database system. *Commun. ACM*, 34:50–63, October 1991.

[35] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou. C-Miner: Mining Block Correlations in Storage Systems. In *Proc. of USENIX FAST*, 2004.

[36] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira. Safe and efficient sharing of persistent objects in thor. In *Proceedings of ACM SIGMOD*, 1996.

[37] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn. Archipelago: trading address space for reliability and security. *SIGARCH Comput. Archit. News*, 36(1):115–124, 2008.

[38] T. Makatos, Y. Klonatos, M. Marazakis, M. D. Flouris, and A. Bilas. Using transparent compression to improve ssd-based i/o caches. In *Proc. of EuroSys*, 2010.

[39] H. M. Mashburn, M. Satyanarayanan, D. Steere, and Y. W. Lee. RVM: Recoverable Virtual Memory, Release 1.3. 1997.

[40] C. Morrey and D. Grunwald. Peabody: the time travelling disk. In *Proc. of IEEE MSST*, 2003.

[41] J. J. Mortensen, L. B. Hansen, and K. W. Jacobsen. Real-space grid implementation of the projector augmented wave method. *Phys. Rev. B*, 71(3):035109, Jan 2005.

[42] E. B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE TSE*, 18(8):657–673, 1992.

[43] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, Lecture Notes in Computer Science, 2002.

[44] R. A. Oldfield, S. Arunagiri, P. J. Teller, S. Seelam, M. R. Varela, R. Riesen, and P. C. Roth. Modeling the Impact of Checkpoints on Next-Generation Systems. *IEEE MSST*, 2007.

[45] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the USENIX Annual Technical Conference*, 1999.

[46] J. S. Plank, M. Beck, and G. Kingsley. Libckpt: transparent checkpointing under Unix. In *Proc. of the USENIX ATC*, January 1995.

[47] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Storage. In *Proc. of USENIX FAST*, 2002.

[48] S. Rhea, R. Cox, and A. Pesterev. Fast, inexpensive content-addressed storage in foundation. In *Proc. of USENIX ATC*, 2008.

[49] M. Rosenblum and J. Ousterhout. The Design And Implementation of a Log-Structured File System. In *Proc. of ACM SOSP*, 1991.

[50] M. Satyanarayanan, H. Mashburn, P. Kumar, D. C. Steer, and J. Kistler. Lightweight Recoverable Virtual Memory. *Proc. of the ACM SOSP*, 1993.

[51] V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas: An efficient, portable persistent store. In *Proceedings of the Intl Workshop on Persistent Object Systems*, September 1992.

[52] V. Sundaram, T. Wood, and P. Shenoy. Efficient Data Migration in Self-managing Storage Systems. In *Proc. of ICAC*, 2005.

[53] The PaX Team. PaX Address Space Layout Randomization (ASLR). *Available online at: http://pax.grsecurity.net/docs/aslr.txt*.

[54] L. Useche, J. Guerra, M. Bhadkamkar, M. Alarcon, and R. Rangaswami. EXCES: EXternal Caching in Energy Saving Storage Systems. In *Proc. of IEEE HPCA*, 2008.

[55] F. Vaughan and A. Dearle. Supporting large persistent stores using conventional hardware. In *In Proc. International Workshop on POS*, 1992.

[56] H. Volos, A. J. Tack, and M. Swift. Mnemosyne: Lightweight persistent memory. In *Proc. of ASPLOS*, 2011.

[57] S. J. White and D. J. DeWitt. A performance study of alternative object faulting and pointer swizzling strategies. In *Proc of VLDB*, 1992.

[58] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proc. of ISMM*, 1992.

[59] B. Zhu, K. Li, and H. Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. *Proc. of USENIX FAST*, Feb 2008.

Rivet: Browser-agnostic Remote Debugging for Web Applications

James Mickens
Microsoft Research

Abstract

Rivet is the first fully-featured, browser-agnostic remote debugger for web applications. Using Rivet, developers can inspect and modify the state of live web pages that are running inside unmodified end-user web browsers. This allows developers to explore real application bugs in the context of the actual machines on which those bugs occur. To make an application Rivet-aware, developers simply add the Rivet JavaScript library to the client-side portion of the application. Later, when a user detects a problem with the application, the user informs Rivet; in turn, Rivet pauses the application and notifies a remote debug server that a debuggable session is available. The server can launch an interactive debugger front-end for a human developer, or use Rivet's live patching mechanism to automatically install a fix on the client or run diagnostics for offline analysis. Experiments show that Rivet imposes negligible overhead during normal application operation. At debug time, Rivet's network footprint is small, and Rivet is computationally fast enough to support non-trivial diagnostics and live patches.

1 Introduction

As an application becomes more complex, it inevitably accumulates more bugs, and a sophisticated debugging framework becomes invaluable for understanding the application's behavior. With modern web browsers providing increasingly powerful programming abstractions like threading [21] and bitmap rendering [9], web pages have become more complex, and thus more difficult to debug. Unfortunately, current debuggers for web applications have several limitations.

Although modern browsers ship with feature-rich JavaScript debuggers, these debuggers can typically only be used to examine pages running on the local browser. This makes it impossible for developers to examine real bugs from pages running on web browsers in the wild. A few browsers do support an interface for remote debugger attachment [4]; however, that interface is tied to a specific browser engine, meaning that the remote debugger cannot be used with other browser types. Given the empirical diversity of the browsers used in the wild [20], and the inability of web developers to dictate which browsers their users will employ, remote debugging frameworks that are tied to a particular browser engine will have poor coverage for exposing bugs in the wild. Indeed, the quirks of individual browsers are important inducers of web application bugs [13, 14]. Thus, an effective remote debugger must be able to inspect pages that run inside arbitrary, unmodified commodity browsers.

1.1 Our Solution: Rivet

In this paper, we introduce Rivet, a new framework for remotely debugging web applications. Rivet leverages JavaScript's built-in reflection capabilities to provide a browser-agnostic debugging system. To use Rivet, an application includes the Rivet JavaScript library. When the page loads, the Rivet library instruments the runtime, tracking the creation of various types of state that the application would otherwise be unable to explicitly enumerate. Later, if the user detects a problem with the web page, she can instruct the page to open a debugging session with a remote developer. The developer-side debugger communicates with the client-side Rivet framework using standard HTTP requests. The debugger is fully-featured and supports standard facilities like breakpoints, stack traces, and dynamic variable modification.

Many laypeople users will not be interested in assisting developers with long debugging sessions. Thus, Rivet also supports an automated diagnostic mode. In this mode, when the user or the application detects a problem, the debug server automatically pushes pre-generated test scripts to the client. The client executes the scripts and sends the results back to the server, where they can be analyzed later. In this fashion, Rivet supports the generation of quick error reports for live application deployments. This mechanism also facilitates the distribution of live patches to client machines.

1.2 Contributions

In summary, Rivet is the first fully-featured, browser-agnostic remote debugger for web applications. Rivet offers several advantages beyond its browser agnosticism. With respect to security, Rivet only allows a remote developer to debug pages from her origin; in contrast, prior (browser-specific) remote debuggers allow a developer to inspect any page in any browser tab. With respect to usability, Rivet does not require end-users to reconfigure their browsers or manage other client-side debugging infrastructure. This is important, since widely deployed applications are primarily used by non-technical laypeople who lack the sophistication to configure network ports or attach their browser to a remote debugging server. Prior remote debuggers require end users to perform such configuration tasks.

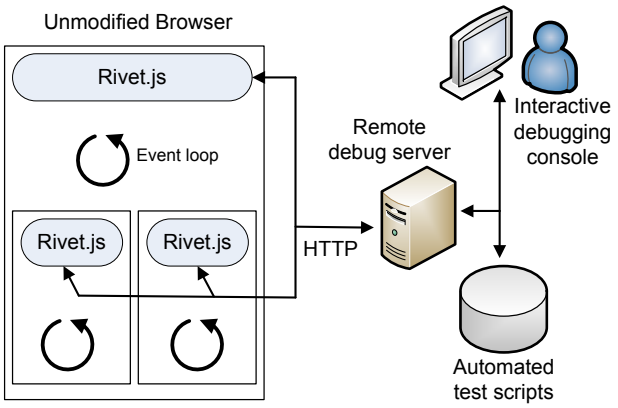


Figure 1: Example of a Rivet web page with three frames.

Rivet hides configuration details from the user by implementing its client-side component totally in JavaScript. This gives the web developer complete control over the client-side debugging settings, while restricting the developer to accessing content from that developer's origin. Rivet's client-side library is partially inspired by Mugshot [14], a logging-and-replay framework for web applications that also leverages JavaScript reflection to work on unmodified browsers. However, as we describe in Section 5, Rivet has three significant advantages over Mugshot. First, Rivet does not require developers to deploy a proxy web server to assist with application introspection; such proxies may be expensive in both cost and performance. Second, Rivet allows developers to explore bugs on the actual end-user browsers in which those bugs occur. This in-situ perspective increases Rivet's bug coverage relative to Mugshot—Mugshot tries to recreate bugs on the developer-side, but it cannot reliably replay bugs which depend on client-side state that resides beneath the JavaScript layer. Third, Rivet performs less runtime interpositioning than Mugshot. Since browser interpositioning can be fragile [13], this makes Rivet more robust than Mugshot.

Our evaluation shows that, through careful design, standard HTTP connections can support a fast, interactive remote debugging protocol. We also show that Rivet introduces negligible client-side overhead during normal application operation. Thus, Rivet is performant enough to ship inside real application deployments.

2 Architecture

Figure 1 depicts the architecture of a Rivet-enabled web application. The client portion of the application runs inside an unmodified commodity browser. The application consists of one or more *event contexts*. An event context is either an iframe or a web worker [21], a constrained type of threading context that we discuss in more detail below. JavaScript is an event-driven language, and each frame or web worker has its own event dispatch loop. Each context also has its own JavaScript namespace.

In a Rivet-enabled application, each event context contains its own copy of the Rivet library. When the application loads, each instance of the library will instrument its local JavaScript runtime. Later, at debug time, these modules will coordinate to pause the application, communicate with the remote debug server, and examine local state on behalf of that server.

On the developer side, the debug server may be connected to a front-end debugging GUI that allows the developer to interactively examine the client portion of the application. Alternatively, the developer can configure the debug server to run automated diagnostics on a misbehaving web page. The latter facility is useful for debugging widely deployed applications in which most users are not professional testers and lack an interest in actively assisting the developer in her debugging efforts. For a popular web page, the debug server may interface with a larger distributed system that collates and analyzes massive sets of error reports [10].

2.1 Exposing Application State

The purpose of the client-side Rivet library is to communicate with a remote debugger and give that debugger a way to inspect the client-side state. In particular, Rivet must allow the debugger to walk the JavaScript object graph that represents the application's state. Each event context has its own JavaScript namespace and possesses one or more partially overlapping object graphs. Below, we describe each type of object graph and how Rivet provides an introspection framework for that graph.

The global namespace forest: JavaScript supports a powerful reflection model. By calling the built-in `Object.getOwnPropertyNames()` method, an application can discover all of the properties defined on an object. [1] JavaScript also makes the global namespace explicitly accessible via the special `window` object. Thus, at debugging time, Rivet can discover the current set of global variables by enumerating the properties of the `window` variable. Rivet can then recursively reflect over the properties of each global variable, mapping the entire object tree rooted at each global.

The Document Object Model (DOM) is a browser-neutral API for exposing browser state to JavaScript programs [24]. With some notable exceptions that we

[1] JavaScript uses a prototype-based object model (§2.5), and `Object.getOwnPropertyNames()` only enumerates properties found directly on the object. Thus, to discover an object's full property list, Rivet must call `Object.getOwnPropertyNames()` on the object itself and on each object in the prototype chain.

```
function closureGenerator(obj){
    var y = 41;
    function closure(){
        //closure() binds to non-local
        //variables obj and y . . .
        return obj.x + y;
    }
    return closure;
}
var globalObj = {x: 1}; //Create object literal
var c = closureGenerator(globalObj);
alert(c()); //Displays 42
```

Figure 2: Example of a JavaScript closure.

```
function closureGenerator(obj){
    var y = 41;
    function closure(){
        return obj.x + y;
    }
    closure.__getScope__ = function(){
        return {"obj": obj, "y": y};
    };
    closure.__evalInScope__ = function(src){
        return eval(src);
    };
    return closure;
}
```

Figure 3: Rivet rewrites the closure from Figure 2 to support exploration by remote debuggers.

describe below, all DOM state is accessible via properties on the `window` object. For example, persistent local storage areas like cookies and DOM storage [25] are exposed via `window.document.cookie` and `window.localStorage`, respectively. For DOM state that is not explicitly accessible in this way, Rivet instruments the runtime to expose the state to remote debuggers.

Closure state: Many JavaScript applications make heavy use of closures. A closure is a function that remembers the values of non-local variables in the enclosing scopes. Figure 2 provides an example of a closure.

The global JavaScript namespace is explicitly bound to the `window` variable. In contrast, a closure namespace is *not* bound to an explicit namespace object. Thus, in Figure 2, once `closureGenerator(1)` returns, neither the returned closure nor any other code can explicitly refer to the bound closure scope—only the closure itself can access the bound variables `obj` and `y`. This is unfortunate from the perspective of debugging, since knowing the values of closure variables can greatly assist the debugging of closure functions. Some closure variables may be aliased by global variables, or reachable from the object graph rooted in a global variable; unfortunately, in non-trivial programs, it is difficult or impossible for a developer (or static code analysis) to identify such aliasing relationships. Furthermore, some closure variables are completely unreachable from the global namespace; in Figure 2, `y` is such a variable.

In JavaScript, all functions, including closures, are first-class objects. To expose closure state to remote debuggers, Rivet uses a JavaScript rewriter to associate each closure with two diagnostic functions that expose the normally hidden scope. Figure 3 shows how Rivet rewrites the closure from Figure 2. The `__getScope__()` function returns an object whose properties reference the closure variables. Note that `__getScope__()` is itself a closure, which lets it access the protected scope of the application-defined closure. When the remote debugger calls `__getScope__()`, it can use standard JavaScript reflection to enumerate the returned object's properties and read the closure variables.

Rivet adds a second diagnostic method to each closure called `__evalInScope__()`. This method allows a remote debugger to dynamically evaluate a piece of JavaScript code within the closure function's scope chain. This allows the debugger to modify the value of a closure variable (or any other variable within the function's scope). As we explain in Section 2.5, `__evalInScope__()` also allows the debugger to dynamically generate a new function that is bound to a closure's hidden scope. This is useful in several scenarios, e.g., when introducing a new debugging version of a closure that automatically produces diagnostic messages.

Note that `__getScope__()` and `__evalInScope__()` are only invoked during a debugging session, so rewritten closures incur no performance penalty during normal application execution. Also note that the closure rewriter can be implemented in a variety of ways. For example, a developer-side IDE can automatically rewrite JavaScript files. Alternatively, the client-side Rivet library can dynamically rewrite script code by interposing on `eval()` and DOM-mediated injection points for generating script code [15].

Importantly, Rivet will not break if some or all of an application's closures are not rewritten. However, Rivet will not be able to dynamically modify those closure functions. This will prevent Rivet from inserting dynamic breakpoints into them (§2.3) or otherwise live-patching them. Rivet can use lexical analysis to identify closures and prevent the developer from issuing forbidden operations on non-rewritten closures. This is an important feature, since an application may import external JavaScript that the developer does not control (and thus cannot ensure will be rewritten).

In JavaScript, calling a function's `toString()` method conveniently returns the source code for that

```
var button = document.getElementById("clicker");
button.onclick = function(){
    alert("DOM 0 handler!");
};
button.addEventListener("onclick",
                        function(){
                          alert("DOM 2 handler!");
                        });
```

Figure 4: Registering GUI callbacks using the DOM 0 model and the DOM 2 model.

```
var req = new XMLHttpRequest();
req.open('GET', "http://foo.com");
req.onReadyStateChange = function(){
    if(req.readyState == 4){
        alert(req.responseText);
    }
}
req.send();
```

Figure 5: An AJAX callback.

function. Thus, Rivet does not need to maintain extra metadata to store function source for subsequent inspection by the remote debugger.

Event handler state: JavaScript applications can define three types of callback functions.

- *Timer callbacks* execute after a specified period of time has elapsed. Callbacks registered via `setTimeout(f, waitMs)` only execute once, whereas callbacks registered via `setInterval(f, periodMs)` fire once every `periodMs` milliseconds.
- To respond to user input, applications define `GUI callbacks` on DOM nodes (each DOM node is the JavaScript representation of an HTML tag in the web page). Modern web browsers support two different registration models for GUI events [7]. The "DOM 0" model allows an application to register at most one handler for a given DOM node/event type pair. Using the "DOM 2" model, an application can register multiple event handlers for a given DOM node/event type pair. A node can simultaneously have a DOM 0 handler and one or more DOM 2 handlers. Figure 4 provides an example of the two registration models.
- To asynchronously fetch new web data, an application creates an `XMLHttpRequest` object and defines an `AJAX callback` for the object [7]. The browser will fire this callback whenever it receives bytes from the remote web server. Figure 5 provides an example.

The client-side Rivet framework must ensure that the remote debugger can enumerate the application-defined callbacks. Thus, when the Rivet library first loads, it wraps `setTimeout()`, `setInterval()`, and the AJAX object in logging code that records the application-defined handlers that are passed through the aforementioned interfaces. The wrapper code is similar to that used by the Mugshot logging and replay service [14]. Later, at debug time, the debugger can access the timer callbacks by examining the special lists `Rivet.timeoutCallbacks`, `Rivet.intervalCallbacks`, and `Rivet.AJAXCallbacks`.

Rivet does not need to do anything to expose DOM 0 handlers to the remote debugger—these handlers are attached to enumerable properties of the DOM nodes. In contrast, DOM 2 handlers are not discoverable by reflection, so Rivet modifies the class definition for DOM nodes, wrapping the `addEventListener()` function that is used to register DOM 2 handlers. The wrapper adds each newly registered handler to a list of functions associated with each DOM node; this list property, called `DOMNode.DOM2handlers`, is a regular field that can be accessed via standard JavaScript reflection.

Web Workers: A web application can launch a concurrent JavaScript activity using the web worker facility [21]. Although a web worker runs in parallel with the spawning context, it has several limitations. Unlike a traditional thread, a web worker does not share the namespace of its parent; instead, the parent and the child exchange pass-by-value strings over the asynchronous `postMessage()` channel. Web workers can generate AJAX requests and register timer callbacks, but they cannot interact with the DOM tree.

An application launches a web worker by creating a new `Worker` object. The application calls the object's `postMessage()` function to send data to the worker; the application receives data from the worker by registering DOM 0 or DOM 2 handlers for the worker's `message` event. At page load time, the Rivet library wraps the `Worker` class in logging code. Similar to Rivet's wrapper code for the `XMLHttpRequest` class, the `Worker` shim allows Rivet to track the extant instances of the class and the event handlers that have been added to those instances. Rivet also shims the AJAX class and the timer callback registration functions inside each worker context.

2.2 Pausing an Application

In the text above, we described how the client-side Rivet library exposes application state to the remote debugger. However, before the debugger can examine this state, the application must reach a *stable point*. A stable

point occurs when Rivet-defined code is running within each client-side event context. Each frame or web worker can only execute a single callback at any given time, so if a Rivet-defined callback is running in a particular context, Rivet can be confident that no other application code is simultaneously running in that context. If Rivet code is running in *all* contexts, then Rivet has effectively paused the execution of all application-defined code. Rivet's client-side portion can then cooperate with the remote debugger to inspect and modify application state in an atomic fashion, without fear of concurrent updates issued by regular application code.

The top-most frame in a Rivet-enabled application is the coordinator for the pausing process. When the user detects a problem with the application, she informs the Rivet library in the root frame, e.g., by clicking a "panic" button that the developer has linked to the special `Rivet.pause()` function. `Rivet.pause()` causes the root frame to send a pause request to the Rivet library in each child frame and web worker; these pause requests are sent using the `postMessage()` JavaScript function. Upon receiving a pause command, a parent recursively forwards the command to its children. When a context with no children receives a pause request, it sends a pause confirmation to its parent and then opens a synchronous AJAX connection to the remote debug server. This synchronous connection allows the Rivet library to freeze the event dispatch process, forcing application-defined callbacks to queue up. A parent with children waits for confirmations from all of its children before notifying its own parent that it is paused and opening its own synchronous connection to the debug server. When the root frame receives pause confirmations from all of its children, it knows that the entire application is paused. The root frame connects to the remote debugger, which can then inspect the state of the application using the introspection facilities described in Section 2.1.

To unpause the application, the remote debugger sends a "terminate" message to each AJAX connection that it has established with the client. Upon receiving this message, each Rivet callback closes its AJAX connection and returns, unlocking the event loop and allowing the application to return to its normal execution mode.

2.3 Breakpoints

Rivet allows developers to dynamically insert breakpoints into a running application's event handlers. To set a breakpoint, the developer must first identify the appropriate event handler using the remote debugger's object enumeration GUI. This interface uses a standard tree view to represent the application's object graphs. Once the developer has found the appropriate function, the remote debugger displays the function's source code by calling the `toString()` of the underlying function object. The developer inserts breakpoints at one or more locations in the source code and then instructs the debugger to update the event handler function on the client side.

```
var lastEvalResult = "Start of breakpoint";
var exprToEval     = "";
while(lastEvalResult){
  exprToEval = Rivet.breakpoint(lastEvalResult);
        //Uses a synchronous AJAX connection
        //to return the result of the prior
        //debugger command and fetch a new one.
  lastEvalResult = eval(exprToEval);
}
```

Figure 6: The Rivet breakpoint implementation.

When the client-side Rivet framework receives the new handler source code, it translates each breakpoint into the code shown in Figure 6. Each breakpoint is just a loop which receives a command from the remote debugger, `eval()`s that command, and sends the result back to the debugger. A breakpoint command might fetch the value of a local variable or reset its value to something new. Communication with the debugger uses a synchronous AJAX connection to ensure that Rivet locks the event loop in the breakpoint's event context.

Once the client-side framework has translated the breakpoints, it dynamically creates a new function representing the instrumented event handler. To do so, Rivet passes the new source code to the standard `eval()` function if the handler to rewrite is not a closure; otherwise, Rivet uses the handler's `__evalInScope__()` function (§2.1). Equipped with the new callback, Rivet then replaces all references to the callback using the techniques we describe later in Section 2.5.

When the debugger unpauses the application, the application will execute normally until it hits a call to `Rivet.breakpoint()`. `Rivet.breakpoint()` must pause the application, but it must not relinquish control of the local event loop while it does so. Thus, `Rivet.breakpoint()` sends standard pause requests to its children, but does not wait for confirmations before marking itself as paused. `Rivet.breakpoint()` also sends a special "breakpoint" message to its parent. This message is recursively forwarded up the event context tree until it reaches the root frame. Upon receiving the message, the root frame pauses the rest of the application. Once the debugger has detected that all of the event contexts are paused, it can interrogate them as necessary.

2.4 Stack Traces

JavaScript does not explicitly expose function call stack frames to application-level code. Nevertheless, when an application hits a breakpoint, Rivet can send a stack

trace to the remote debugger. Rivet defines two kinds of stack traces. A lightweight stack trace reports the function name and current line number for each active call frame. Even though Rivet does not have explicit access to the call stack, it can access function names and line numbers by intentionally generating an exception within a try/catch block, and extracting stack trace information from the native `Exception` object that the browser generates. This is the same technique used by the stacktrace.js library [23], and it works robustly across all modern browsers.

Lightweight stack traces do not expose actual call frames to Rivet. Thus, although Rivet breakpoints can use an `eval()` loop to provide read/write access to local variables in the topmost stack frame, Rivet cannot use lightweight stack traces to introspect variables in call frames that are lower in the stack. Rivet can provide heavyweight stack traces that provide such functionality, but to do so, Rivet must rewrite *all* functions so that they update a stack of function objects upon each call and return. Furthermore, each function must define the `__getScope__()` and `__evalInScope__()` that Rivet uses to introspect upon closure state (§2.1).

Both closure rewriting and heavyweight stack rewriting can be done statically, before application deployment. However, the function code that supports heavyweight stack traces can be dynamically swapped in at debug time. This allows the application to avoid the bookkeeping overhead during normal operation. Note that rewritten closure code should *always* be used, lest Rivet miss the creation of a closure scope and be unable to expose the bound variables to the remote debugger.

2.5 Generic Live Patching

Rivet defines a patch as a single JavaScript function which Rivet will evaluate in the global scope. The patch can access and modify the objects reachable from the global variables without assistance from Rivet. The patch accesses closure scopes through the `__getScope__()` function that Rivet adds to each closure. The patch can also access normally non-enumerable event handlers using data structures like `Rivet.timeoutCallbacks` and `DOMNode.DOM2handlers`.

To make patches easier to write, Rivet defines three convenience functions for developers. The first, called `Rivet.overwrite(oldObj, newObj)`, instructs Rivet to copy all of `newObj`'s values into `oldObj`. If neither `newObj` nor `oldObj` are a function, the overwriting process is trivial—since JavaScript objects are just dictionaries mapping property names to property values, Rivet can overwrite an object in place by deleting all of its old properties and assigning it all of `newObj`'s properties. If `newObj` or `oldObj` is a function (or another native code object like a regular expression), Rivet must use a different approach, since these objects are bound to opaque internal browser state that is not easily transferred to other objects. Thus, to overwrite a native object, Rivet must traverse the application's entire object graph and, for each object, replace any references to `oldObj` with references to `newObj`. Rivet uses standard techniques from garbage collection [12] to avoid infinite recursion when the object graph contains cycles.

```
//Define a constructor function for class X.
function X(){
    this.prop1 = 'one';
}
X.prototype.prop2 = 'two';

var x1 = new X();
var x2 = new X();
alert(x1.prop1); //'one': defined in constructor
alert(x1.prop2); //'two': defined by prototype

x1.prop1 = 'changed';
alert(x1.prop1 == x2.prop1);  //False

//Changing the value for a prototype
//property changes the value in all
//instances of that class *unless* an
//instance has explicitly redefined the
//property.
X.prototype.prop2 = 'cat';
alert(x1.prop2 == 'cat');  //True
x2.prop2 = 'dog';
alert(x2.prop2 == 'cat');  //False
```

Figure 7: Example of prototype-based inheritance.

The second convenience function, called `Rivet.redefineClass(oldCtor, makeNewVersion, newCtor)`, is useful for updating all instance objects of a modified class definition. Whereas C++ and Java implement classes using types, JavaScript implements classes using *prototype objects* [7]. Any JavaScript function can serve as a constructor if its invocation is preceded by the `new` operator; any references to `this` inside the constructor invocation will refer to the new object that is returned. Furthermore, by defining a `prototype` property for a function, an application defines an exemplar object which provides default properties for any instance of that constructor's objects. Figure 7 provides a simple example of JavaScript's prototype-based classes.

When a patch invokes `Rivet.redefineClass(oldCtor, makeNewVersion, newCtor)`, Rivet does the following:

- Rivet finds all instances of `oldCtor`'s class, i.e., all objects whose `__proto__` field is `oldCtor.prototype`.
- For each instance `obj`, Rivet calls `makeNewVersion(obj)`, creating a new

version of the object using application-specific logic. Rivet then update `oldObj` in-place with the contents of `newObj`.

- Finally, Rivet uses `Rivet.overwrite(oldCtor,newCtor)` to replace stale references to `oldCtor` with references to `newCtor`.

Applications that desire finer-grained control over their patching semantics can invoke `Rivet.traverseObjectGraph(callback)`, specifying a function that Rivet will call upon each object in the application.

2.6 Debugging Scenarios

We envision that Rivet will be used in two basic scenarios. In the beta testing scenario, a small number of professional testers or motivated volunteers interact with an unpolished version of an application. These testers are not developers themselves, but they are willing to start and stop their application sessions and actively help the developers to debug any problems that arise. In this scenario, when the beta tester encounters a problem, she initiates a full-blown debugging interaction with a remote developer. The developer starts the debugging GUI and engages in an iterative process with the beta tester and the remote application, exploring and modifying the program state in various ways. Using Rivet's live patching mechanism, the developer can dynamically add a chat interface to the live page; this enables a real-time dialogue between the tester and the developer without requiring the tester to configure a separate out-of-band communication mechanism.

For a widely deployed application, the typical layperson user may not want to assist with a remote debugging session. In these situations, the application can still contain a "panic" button. However, when this button is pressed, the application does not initiate a remote debugging session with a human developer—instead, the web page connects to an automated debug server which runs a set of predefined diagnostics on the page. Each diagnostic is implemented as a Rivet patch which simply runs a test on the client-side state and uploads the result to a server. For example, a patch might generate an application-level core dump, serializing the DOM tree and application heap and sending it to the server for further analysis. As another example, a diagnostic could run integrity checks over the application's cookies and other persistent client-side data. These kinds of automatic tests require no assistance from the end user, but provide invaluable debugging information to application developers. These tests can also be silently initiated by the Rivet library, e.g., in response to catching an unexpected exception. In some applications, such automatic diagnostics may be preferable to having a user-triggered "panic" button.

2.7 Implementation

Our Rivet prototype consists of a client-side JavaScript library and a developer-side debugging framework that contains a JavaScript rewriter, a debug server, and a front-end debugging GUI. After minification (i.e, the removal of comments and extraneous whitespace), the client-side library is less than 29 KB of source code; thus, it adds negligible cost to the intrinsic download penalty for a modern web application that contains hundreds of KB of JavaScript, CSS, and images [18]. The closure rewriter and the debug server are both written in Python. Before deploying an application, the developer passes its JavaScript code through the rewriter, which instruments closures as described in Section 2.1. After deploying the application, the debug server listens for debugging requests from remote web pages. If the server is configured to run in auto-response mode, it will run a pre-selected diagnostic script on the remote application and store the results in a database. Otherwise, if the server is set to interactive mode, it will open a front-end debugging GUI which the developer can use to inspect the remote application in real time. The front-end is just a web page that communicates with the debug server via HTTP. The server acts as a relay between the front-end and the remote application, sending debugger commands to the page and sending client-generated results back to the front-end.

Our current prototype implements all of the features described in this section except for heavyweight stack traces (§2.4). We are currently building a rewriting engine to support this facility. The client-side Rivet library has been tested extensively on Firefox, Safari, IE, and Chrome, and it works robustly on those browsers.

3 Privacy Concerns

Rivet exposes all of a web page's state to a remote debugger. This state might include personal information like passwords, emails, e-commerce shopping carts, and so on. Applications can use HTTPS to secure the connection between the web browser and the debug server; by doing so, a client can easily verify the identity of the remote principal that is inspecting local state. HTTPS also prevents arbitrary network snoopers from inspecting client data. However, it does not constrain the remote debugger's ability to enumerate and modify client-side state.

Rivet does not grant fundamentally new inspection powers to developers, since applications do not need Rivet to take advantage of JavaScript's powerful reflection capabilities. For example, there are many preexisting JavaScript libraries that analyze user activity and page state to determine which ads a user clicks or which parts of a page are accessed the most. Also, much of the private client-side information is persistently stored in

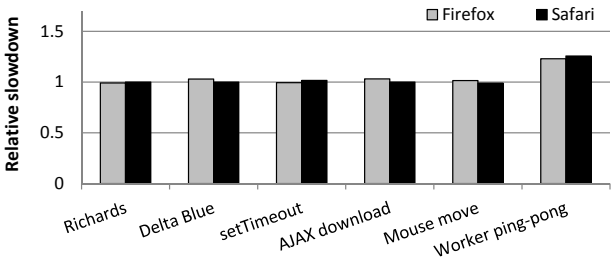


Figure 8: Microbenchmarks: Relative slowdown of Rivet-enabled versions. A slowdown factor of 1 indicates no slowdown.

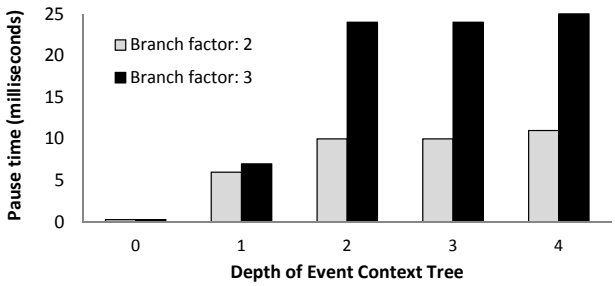


Figure 9: Time needed to pause a nested frame tree with no event loop contention.

server-side databases, with the client application treated as an ephemeral cache. Thus, developers can already inspect much of the user's private data without interrogating the client portion of the application.

4 Evaluation

In this section, we investigate four questions. First, how much overhead does Rivet add to an application during normal usage, i.e., when the application is *not* being debugged? Second, how long does it take for Rivet to pause an application? Third, how large are the messages that are exchanged between the client-side Rivet library and the debug server? Finally, how long does it take for Rivet to patch a live application or run an automated diagnostic? Using synthetic benchmarks and real applications, we show that Rivet adds negligible overhead during normal operation. Furthermore, at debug time, Rivet is fast enough to support interactive remote debugging sessions.

To test Rivet's performance on web applications in the wild, we loaded those applications through a rewriting web proxy. The proxy added the Rivet JavaScript library to each HTML file and web worker; it also passed all of the JavaScript code through Rivet's closure rewriter. Using this proxy, we could test Rivet's performance on live web applications without requiring control over the servers that actually deliver each application's content. In all graphs, each result is the average of ten trials. Each trial was run on a Dell workstation with dual 3.20 GHz processors, 4 GB of RAM, and the Windows 7 operating system. For the experiments in this section, we used Firefox 6.0.2 and Safari 5.1 to load Rivet-enabled applications. For graphs using only a single browser, Firefox was used.

4.1 Microbenchmarks

Computational slowdown: Figure 8 shows the relative performance slowdown for several Rivet-enabled microbenchmarks. We used the following benchmarks:

- `Richards` and `Delta Blue` are CPU-bound benchmarks from Google's V8 test suite.
- `setTimeout` measures how quickly the browser can dispatch timer callbacks that have a delay time of zero.
- The `AJAX download` test downloads a 100 KB file from a local web server on the same machine as the browser. Using a local web server eliminates the impact of external network conditions and isolates Rivet's impact on client-side AJAX handling.
- In the `Mouse move` test, the web page registers a null callback for mouse movement events. A human user then quickly moves the mouse cursor back and forth across the screen for ten seconds. The output of the benchmark is how many times the browser invoked the GUI callback.
- The `Worker ping-pong` test examines how quickly a web worker and a parent frame can send `postMessages()` to each other.

With the exception of the `Worker ping-pong` test, the performance of the Rivet-enabled benchmarks was statistically indistinguishable from that of the standard benchmarks. There are several reasons for Rivet's low overhead. First, the Rivet initialization code that runs during page load is extremely fast. This code merely needs to interpose on some class definitions and global methods, and this process takes less than one millisecond (which is the best resolution of JavaScript timestamps on modern browsers). Post-initialization, Rivet's bookkeeping code is also extremely fast. For example, the overhead of tracking `setTimeout()` callbacks is five lines of bookkeeping code and two extra function calls (one made from the wrapped `setTimeout()` to the native version, and one made from the wrapped callback to the real, application-supplied function). Rivet's overhead for tracking closure state is also low. As shown in Figure 3, rewritten closures do not execute any Rivet code during normal usage—the `__getScope__()` and `__evalInScope__()` functions are only invoked by the remote developer at debug time.

Figure 8 shows that Rivet makes the `Worker ping-pong` test roughly 25% slower. The primary reason is that browsers dispatch worker messages with much higher throughput than the rate at which they dispatch zero-delay timer callbacks or mouse events. As a concrete example, on our dual-core test machine, Firefox 6 could issue roughly 44 mouse events a second and 163 zero-delay timer callbacks a second, but over 16,000 ping-pong exchanges per second. Thus, Rivet's `postMessage()` overheads are comparatively larger. In a Rivet application, a single round of ping-pong involves three wrapped function calls: the `postMessage()` on the worker object in the parent context, the message callback in the worker context, and the message callback in the parent context that handles the worker's response.

Pause latency: A web application consists of a tree of event contexts. Rivet pauses an application by disseminating pause requests across this tree and waiting for child contexts to acknowledge that they are paused; the entire application is paused once the root frame has received pause confirmations from all of its children. Figure 9 depicts the pause latency as perceived by the root frame for event trees of various depths and branching factors. None of the event contexts defined any application-level handlers, so Rivet's pause handlers could execute as quickly as possible. Thus, the results in Figure 9 represent the lowest possible pause latencies.

Figure 9 shows that the intrinsic pause delay is extremely small. Even for an unrealistically dense tree with a branching factor of three and a depth of four, the pause process only takes 25 milliseconds. Of course, fully pausing an application can take an unbounded amount of time if an event handler contains code that runs for an extremely long time. However, we expect such situations to be rare, at least for handlers defined in frames, since developers know that long-running, non-yielding computations may freeze the browser's UI. Web workers were designed specifically so that applications could execute such computations without affecting the user interface; thus, web workers are a more likely source of long-running event handlers. However, in these situations, developers can explicitly insert Rivet breakpoints in worker code to place an upper-bound on an application's pause time.

Rivet's current pausing scheme guarantees that *all* contexts are in non-volatile states when debugging occurs. Rivet could trade this guarantee for the ability to diagnose hung contexts. At application load time, Rivet could create an invisible master frame that resided atop the context hierarchy. This master frame, controlled by Rivet, would always be guaranteed to be live. By coordinating with the Rivet libraries in each descendant event context, the master frame could build a list of all such contexts. Using this list, the master frame could send pause requests to each descendent context directly (instead of sending requests down the context tree). After a timeout, the master frame would declare any silent contexts to be hung. At that point, the master frame would yield to the remote debugger. The remote debugger could inspect both paused contexts and hung contexts as normal. However, *all* contexts would be in potentially volatile states, since a hung event handler could unhang at any moment and mutate some context's state before relinquishing the processor.

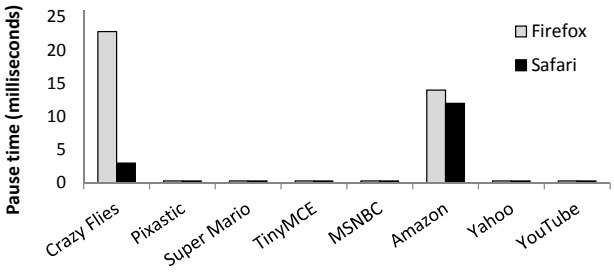


Figure 10: Time needed to pause several applications.

4.2 Macrobenchmarks

Pause latency: Figure 10 depicts the time needed to pause eight real applications. `Crazy Flies` simulates an insect swarm, using a web worker to calculate insect movements and a frame to depict an animation of the simulated insects' movements. `Pixastic` is an image manipulation program that supports standard operations like hue adjustment, noise removal, and edge detection. `Super Mario` is a JavaScript port of the popular 8-bit Nintendo game. `TinyMCE` is a full-featured word processor. The remaining web pages (`MSNBC`, `Amazon`, `Yahoo`, and `YouTube`) are the start pages for the associated web portals.

Figure 10 shows that in all but two cases, application pause times are essentially zero. This is because in most applications, the depth of the event context tree is either zero or one, and in each event context, there is little contention for the local event loop. The former means that the spanning tree for pause dissemination messages is small; the latter means that the Rivet library in each context does not need to wait long before it can grab the local event loop and pause the context. Thus, pausing is often very fast. Both `Crazy Flies` and `Amazon` had event context trees of depth one (`Crazy Flies` has a web worker, and `Amazon` has several frames). However, in both cases, pause times were less than 25 milliseconds.

Message sizes: Figure 11 depicts the size of the messages that Rivet generated during a debugging session for the `Yahoo` page. At the start of the session, the debug-

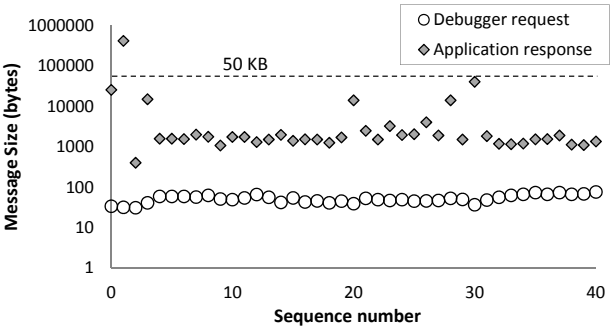


Figure 11: Size of messages exchanged between the remote debugger and the client application.

ger automatically fetched the list of global variables and information about the children of the root `<html>` DOM node; this allowed the debugger GUI to populate the initial tree view elements that a human developer uses to explore the DOM tree and the object graph rooted by the `window` object. After this initialization completed, a human developer used the debugger GUI to explore various DOM nodes and application-defined objects.

Control messages from the remote debugger to the paused client application were extremely small, having an average size of 53 bytes and a maximum size of 74 bytes. The primary content of each control message was a list of property strings that indicated the path to the object whose contents should be returned. Compared to debugger requests, client responses were more variable in size, since objects had widely varying property counts. However, these messages were also small. The average client response was 13 KB, and all but one message was smaller than 50 KB. The outlier was the initial fetch of the global variable list and the properties of the associated objects. The browser defines over 200 built-in global variables, and a given application often adds ten or twenty more. Thus, the number of properties to fetch for the initial object view is often much larger than subsequent ones. Also note that for each function, the Rivet GUI fetches the associated source code by calling that function's `toString()`. This source code comprises the bulk of the initial view fetch.

Diagnostics: Once a web page is paused, a developer can run diagnostics on it or install a live patch. Rivet allows arbitrary dynamic code to be run on or inserted into the client-side. In this section, we discuss a few concrete examples of what a diagnostic might look like.

`Rivet.traverseObjectGraph(f)` allows the debugger to evaluate the function `f` over every object in an application. Thus, `Rivet.traverseObjectGraph()` is a useful foundation for many types of whole-application diag-

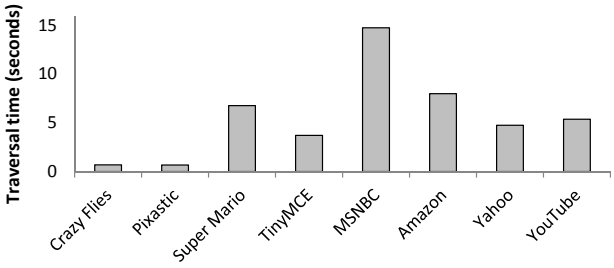
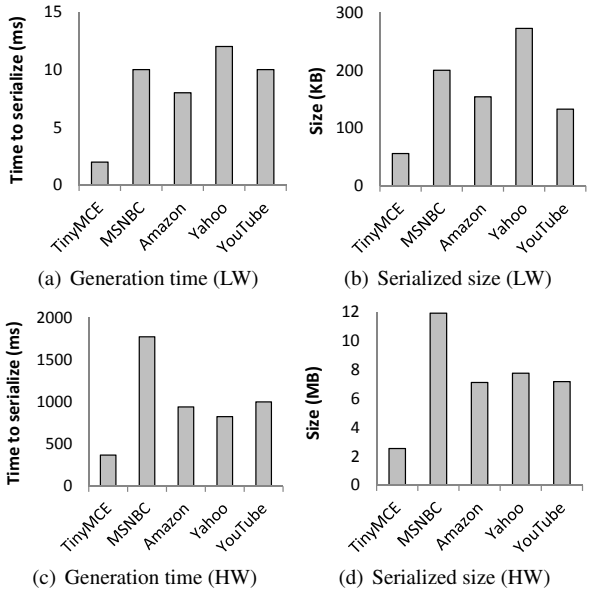


Figure 12: Time needed to visit every application object.



(a) Generation time (LW) (b) Serialized size (LW)

(c) Generation time (HW) (d) Serialized size (HW)

Figure 13: Computational overheads for lightweight (LW) and heavyweight (HW) DOM tree serialization.

nostics. Figure 12 shows how long it takes Rivet to invoke a null function upon each object belonging to a particular application. This time is roughly linear in the number of objects that the application contains. In most cases, a full traversal only takes a few seconds. This means that a Rivet page supporting non-interactive, full graph diagnostics will only have to ask the user to keep a malfunctioning page open for a few seconds before the user can close it. Note that `MSNBC` has the largest traversal time (almost 15 seconds) because it has both a complex graph of application-defined JavaScript objects, and a complex, deep DOM tree. Although applications like `TinyMCE` and `Super Mario` also have large non-DOM object graphs, their DOM trees are comparatively simpler, leading to smaller traversal times.

Visual layout bugs are a common source of frustration in web applications [13]. Thus, developers will often be uninterested in viewing an application's entire object graph—instead, developers will be specifically

interested in the application's DOM tree. Rivet provides built-in support for two types of DOM serialization. Lightweight serialization simply returns a string representing the page's HTML—Rivet easily generates this string by reading the `innerHTML` property of the page's `<html>` DOM node. The `innerHTML` string only reflects simple DOM properties like tag names and node ids, so it does not capture more complex application-defined properties like DOM 2 event handlers. Rivet's second type of serialization, which we call heavyweight serialization, does capture these properties. Heavyweight serialization essentially generates a serialized debugger view for the DOM tree. However, instead of generating view data for nested objects on demand, this data is recursively gathered in one sweep for analysis by the debug server.

Figure 13 depicts the computational requirements for lightweight and heavyweight DOM serialization. We ran the experiments on the five applications with the largest DOM trees. As expected, lightweight serialization was faster and generated smaller outputs than heavyweight serialization. The implementation of `innerHTML` resides completely within the native code of the browser, so reading the value only required a few milliseconds (Figure 13(a)). The resulting serialized strings were between 56 KB and 273 KB in size (Figure 13(b)). In contrast, generating the heavyweight serialization string was much slower, since the DOM tree traversal took place in application-level JavaScript code instead of browser-level C++ code. Figure 13(c) shows that heavyweight serialization took between 370 milliseconds and 1773 milliseconds. The serialization strings were also much larger, with the most complex sites like `MSNBC` and `Yahoo` having serialized DOM trees of 7–12 MB. On a modern DSL connection, such trees would take a few seconds to upload to the debug server. However, like the other client messages, we expect the serialized trees to be amenable to compression.

Live patching: Rivet supports live patching in addition to remote debugging. The installation speed of a live patch depends on the nature of that specific patch. For example, we created a patch that dynamically updated the `MSNBC` page to call an input sanitizer [15] whenever the user entered data into a text box; the input sanitizer stripped any dangerous HTML characters from the input, preventing cross-site scripting attacks [16]. The patch code used the built-in `document.getElementByTagName()` method to fetch all of the DOM nodes corresponding to text inputs. For each node, the patch installed a new handler for the `onchange` event that invoked the sanitizer.

Since `document.getElementByTagName()` is implemented by native browser code, this patch took only 11 milliseconds to execute. However, patches that are primarily implemented by application-level JavaScript will take longer to run. For example, we wrote a patch for `TinyMCE` that changed the definition of a class that represents URLs. The patch used the `Rivet.redefineClass()` method (§2.5) to perform the necessary modifications to the object graph. The resulting patch took almost eight seconds to install. This is because `Rivet.redefineClass()` has to make expensive application-level traversals through the object graph to replace stale object references.

Bug coverage: We have successfully used Rivet to explore known bugs in several in-house web applications; these bugs were caused by incorrect JavaScript code in the applications. Rivet can also explore the effects of bugs that arise from client-side configuration state that is not directly accessible to the JavaScript interpreter. For example, we used Rivet to reproduce a problem with the Firebug [6] plugin for Firefox, whereby enabling the plugin caused a severe performance decrease for the built-in JavaScript `eval()` function [5]. Mugshot [14], a JavaScript-level diagnostic framework like Rivet, cannot reliably examine bugs that involve client-side state beneath the JavaScript layer (§5).

5 Related Work

All modern browsers have built-in JavaScript debuggers. In most cases, these debuggers are only useful for examining the state of web pages running inside the local browser. The WebKit engine [22] used by Safari and Chrome does support remote debugging [4, 11, 17]. However, this debugging framework has three disadvantages. First, it only works for pages running inside Safari or Chrome. Second, the browser-side portion of the debugger must be manually configured by the end-user; for example, the user must decide which debug server should be allowed to connect to the local browser. Third, the debugging framework allows remote developers to inspect *any* page running on the remote browser, not just the ones that were created by the developer. In contrast, Rivet works on all browsers, requires no configuration work from end-users, and prevents a remote developer from inspecting pages that do not reside in her origin. Rivet also provides rich support for automated diagnostics, live patches, and real-time communication between end-users and remote developers.

There are browser-specific extensions for Firefox [1, 8] and the Android mobile browser [3] that add remote debugging facilities. Rivet has similar advantages over these systems—lack of end-user configuration activity, browser-agnosticism, and so on. However, because Rivet

runs at the application level instead of inside the browser, Rivet cannot provide some of the low-level services that in-browser debuggers can provide. For example, Rivet cannot clear the browser cache or query the JavaScript garbage collector.

The JSConsole tool [19] provides remote access to a page's JavaScript logging console. JSConsole redefines the JavaScript `console` variable, replacing it with an object that implements the standard `console` interface but also accepts commands from a remote debugger. This allows a developer to inspect log messages generated by the application. The developer can also evaluate new JavaScript expressions within the context of the remote application. Like Rivet, JSConsole is browser-agnostic. However, compared to the debugging interface provided by Rivet or an in-browser debugger, the console interface is very limited. For example, JSConsole provides no way to set breakpoints, inspect closure state, or enumerate timer callbacks.

Mugshot [14] is a logging and replay framework for JavaScript web applications. Using Mugshot, a user who encounters a buggy application run can upload a log of the application's nondeterministic events to a remote developer. The developer can then replay the buggy execution run on a local browser, using the local browser's debugger to inspect the application's state.

Mugshot shares Rivet's goal of running on unmodified commodity browsers, and Rivet employs some of Mugshot's introspection techniques to instrument event contexts. However, Rivet interposes on fewer browser interfaces; this makes Rivet more robust and easier to maintain, since browser interpositioning is challenging to get correct [13, 14]. Mugshot also requires developers to deploy a special replay proxy that sits between the end user and the application web server. This proxy records the content and delivery order of client-requested information so that load order and load content can be faithfully recreated at replay time. Deploying this proxy may involve non-trivial effort, particularly if the application fetches content from external origins, since that content must be mirrored by the application's home servers so that it can be fetched through (and recorded by) the replay server. Rivet requires no such infrastructure. Rivet also has the advantage that it can examine application bugs in situ instead of having to transfer a log to the developer machine and recreate the application's state inside the developer's browser. This recreation process is somewhat fragile; for example, replay may lack fidelity if the developer does not select the same type of browser that the user has, or if the bug depends on client-side configuration state like DLLs that the client-side Mugshot library cannot see (and thus cannot describe to the remote developer). In contrast, Rivet runs inside the buggy application itself, and does not require state transferral or recreation. Rivet's interactive debugging mode allows developers to receive descriptions of local configuration state from the end-user.

Ksplice [2] is a system for live patching the Linux kernel. A Ksplice patch updates the kernel at the granularity of a function—to replace an old function, Ksplice adds the new function code to the kernel's address space and then inserts a jump instruction to the new code at the start of the old function. Rivet can perform similar tricks using JavaScript's built-in facilities for object reflection. Similar to Rivet's notion of a stable point, Ksplice defines a quiescent kernel function as one that is not on the call stack of any thread; only quiescent functions may be patched. If a Ksplice patch changes the layout of data structures, the developer must provide code to change these structures. Ksplice ensures that the execution of this code takes place within the same atomic transaction that updates functions. This process is similar to Rivet's mechanism for updating object definitions.

6 Conclusions

Rivet is the first browser-agnostic remote debugger for web applications. Rivet works on unmodified commodity browsers, taking advantage of JavaScript's intrinsic capabilities for dynamic object reflection and modification. Rivet's client-side consists of a JavaScript library; this library adds debugging hooks to the application and communicates with a remote debug server. Upon application failure, the remote server can initiate an interactive debugging session with the remote developer, or run automatic diagnostic scripts that produce data for offline analysis.

Experiments show that Rivet adds negligible overhead during standard application operation. At debug time, Rivet can pause an application in tens of milliseconds; subsequent debugging traffic between the application and the debug server is quite small, with debugger-to-application messages being 53 bytes on average, and application-to-debugger messages being 13 KB on average. Experiments also show that Rivet can efficiently support non-trivial diagnostics and live patches.

References

[1] ActiveState Software. Debugging JavaScript: Komodo 4.4 Documentation. `http://docs.activestate.com/komodo/4.4/debugjs.html`, 2011.

[2] J. Arnold and M. F. Kaashoek. Ksplice: Automatic Rebootless Kernel Updates. In *Proceedings of EuroSys*, Nuremberg, Germany, April 2009.

[3] H. Correia. Remote JavaScript Debugging on Android. `http://www.sencha.com/blog/`

remote-javascript-debugging, November 5 2010.

[4] P. Feldman. WebKit Remote Debugging. http://www.webkit.org/blog/1620/webkit-remote-debugging/, May 9 2011.

[5] Firebug development site. Issue 732: eval() extremely slow when Firebug enabled. http://code.google.com/p/fbug/issues/detail?id=732, November 28 2011.

[6] Firebug: Web Development Evolved. http://getfirebug.com/, 2011.

[7] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, Inc., 5th edition, 2006.

[8] A. Fritze. JSSh: A TCP/IP JavaScript Shell Server for Mozilla. http://crozilla.com/bits_and_pieces/jssh/, 2009.

[9] S. Fulton and J. Fulton. *HTML5 Canvas*. O'Reilly Media, Inc., 1st edition, 2011.

[10] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *Proceedings of SOSP*, Big Sky, Montana, October 2009.

[11] Google. Chrome Developer Tools: Remote Debugging. http://code.google.com/chrome/devtools/docs/remote-debugging.html, 2011.

[12] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[13] J. Mickens and M. Dhawan. Atlantis: Robust, Extensible Execution Environments for Web Applications. In *Proceedings of SOSP*, Lisbon, Portugal, October 2011.

[14] J. Mickens, J. Howell, and J. Elson. Mugshot: Deterministic Capture and Replay for JavaScript Applications. In *Proceedings of NSDI*, San Jose, CA, April 2010.

[15] M. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. Draft specification, January 15, 2008.

[16] National Vulnerability Database. CVE-2010-2301, 2010. Cross-site scripting vulnerability: innerHTML.

[17] PhoneGap. Weinre: Web Inspector Remote. http://phonegap.github.com/weinre/, 2011.

[18] S. Ramachandran. Web metrics: Size and number of resources. Google. http://code.google.com/speed/articles/web-metrics.html, May 26 2010.

[19] R. Sharp. Remotely debug a mobile web app. http://www.jsconsole.com/remote-debugging.html, 2011.

[20] StatCounter. Global Stats: Top 5 Browsers from Sept. 2010 to Sept. 2011. http://gs.statcounter.com/, 2011.

[21] Web Hypertext Application Technology Working Group (WHATWG). Web Applications 1.0 (Living Standard): Web workers. http://www.whatwg.org/specs/web-apps/current-work/complete/workers.html, September 30 2011.

[22] WebKit Open Source Project. http://www.webkit.org/, 2011.

[23] E. Wendelin. stacktrace.js: A framework-agnostic, micro-library for getting stack traces in all web browsers. http://stacktracejs.org/, 2011.

[24] World Wide Web Consortium. Document Object Model (DOM). http://www.w3.org/DOM/, January 19 2005.

[25] World Wide Web Consortium: Web Apps Working Group. Web Storage: W3C Working Draft. http://www.w3.org/TR/2009/WD-webstorage-20091029, October 29 2009.

Wimpy Nodes with 10GbE: Leveraging One-Sided Operations in Soft-RDMA to Boost Memcached

Patrick Stuedi
IBM Research, Zurich
stu@zurich.ibm.com

Animesh Trivedi
IBM Research, Zurich
atr@zurich.ibm.com

Bernard Metzler
IBM Research, Zurich
bmt@zurich.ibm.com

Abstract

Recently, various key/value stores have been proposed targeting clusters built from low-power CPUs. The typical network configuration is that the nodes in those clusters are connected using 1 Gigabit Ethernet. During the last couple of years, 10 Gigabit Ethernet has become commodity and is increasingly used within the data centers providing cloud computing services. The boost in network link speed, however, poses a challenge to the cluster nodes because filling the network link can be a CPU-intensive task. In particular for CPUs running in low-power mode, it is therefore important to spend CPU cycles used for networking as efficiently as possible. In this paper, we propose a modified Memcached architecture to leverage the one-side semantics of RDMA. We show how the modified Memcached is more CPU efficient and can serve up to 20% more GET operations than the standard Memcached implementation on low-power CPUs. While RDMA is a networking technology typically associated with specialized hardware, our solution uses soft-RDMA which runs on standard Ethernet and does not require special hardware.

1 Introduction

The ever increasing amount of data stored and processed in today's data centers poses huge challenges not only to the data processing itself but also in terms of power consumption. To be able to move from petascale computing to exascale in the near future, we need to improve the power efficiency of data centers substantially. Power consumption plays a key role in how data centers are built and where they are located. Facebook, Microsoft and Google have all recently built data centers in regions with cold climates, leveraging the cold temperature for cooling. Power consumption is also considered when choosing the hardware for data centers and supercomputers. For instance, the IBM Blue Gene supercomputer[4] is composed of energy-efficient CPUs running at a lower clock speed than traditional CPUs used in desktop computers.

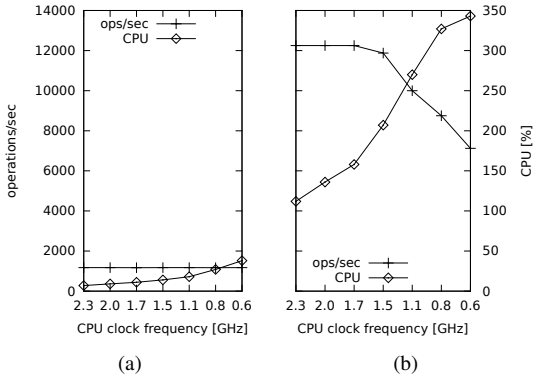


Figure 1: Performance and CPU consumption of a Memcached server attached to (a) 1GbE and (b) 10GbE for different CPU clock rates on a Intel Xeon E5345 4 core CPU

While clusters of low-power CPUs might not necessarily be the right choice for every workload [11], it has been shown that such "wimpy" nodes can indeed be very efficient for implementing key/value stores [5, 6], especially if nodes within the cluster comprise large numbers of cores. The key insight is that even a slow CPU can provide sufficient performance to implement simple PUT/GET operations, given that those operations are typically more network- than CPU-intensive. The common assumption here is that the network used to interconnect the wimpy nodes in the cluster is typically a 1 Gigabit Ethernet network. In the past couple of years, however, 10 Gigabit Ethernet has become commodity and is already used widely within data centers.

Unfortunately, the task of serving PUT/GET requests on top of a 10 Gibabit link imposes a much higher load on the host CPU, a problem for low-power CPUs running at a relatively low clock frequency. Figure 1 shows

Value Size	1K	10K	100K
Total CPU cycles	46K	84K	289K
Networking	35%	42.8%	58%
User Space	5%	3.2%	1.1%
Remaining	60%	54%	40.9%

Figure 2: Breakdown of CPU cycles used by Memcached per GET operation at 1.1Ghz CPU clock frequency

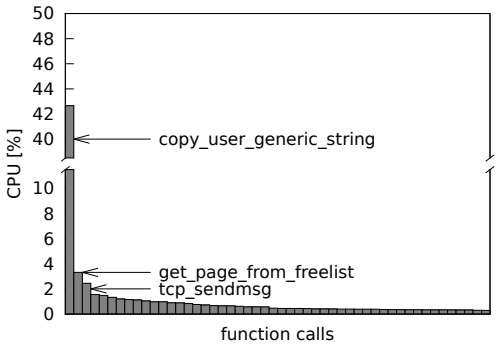


Figure 3: Distribution of the CPU consumption across function calls of the Memcached process serving 100K bytes requests

the performance and CPU consumption of a Memcached server for different CPU clock frequencies in a configuration with 1GbE and 10GbE. As the clock speed decreases in the 1GbE case, the performance stays constant, whereas the CPU consumption slightly increases. In the same experiment but using 10GbE, the performance drops significantly (still higher than what could be achieved with 1GbE though), whereas the CPU consumption reaches close to 350% (more than 3 cores fully loaded).

To get a sense of how those CPU cycles are consumed we have used OProfile [2] to divide the cycles into three classes: "network" – TCP/IP and socket processing, "application" – Memcached user space processing, and "remaining" – all other cycles executed by the operating system on behalf of the Memcached user process, e.g., context switching. Table 2 shows the cycles used by Memcached for each of those classes during the execution of a single GET operation at 1.1Ghz CPU clock frequency. For smaller size GET operations (first column, Table 2) most of the cycles are used up in the "remaining" class. For bigger value sizes (last column, Table 2), the bulk of the CPU cycles are used up inside the network stack. Figure 3 shows exactly how the network related cycles are distributed in a experiment with GET requests for 100K Bytes key/value pairs. A large fraction of the cycles are used to copy data from user space to kernel during socket read/write calls as well as during the actual data transmission.

In this paper, we propose modifications to Memcached to leverage one-sided operations in RDMA. With those modifications in place, Memcached/RDMA uses fewer CPU cycles than the unmodified Memcached due to copy avoidance, less context switching and removal of server-side request parsing. As a result, this allows for 20% more operations to be handled by Memcached per second. While RDMA is a network technology typically associated with specialized hardware, our solution uses soft-RDMA which runs on standard Ethernet and does not require special hardware.

2 Background

To provide low data access latencies, many key/value stores attempt to keep as much of their data in memory. Memcached[1] is a popular key/value store serving mostly as a cache to avoid disk I/O in large-scale web applications, thus keeping all of its data in main memory at any point in time. Remote Direct Memory Access (RDMA) is a networking concept providing CPU efficient low-latency read/write operations on remote main memory. Therefore, using RDMA principles to read, and possibly write, key/value pairs in Memcached seems to be an attractive opportunity, in particular on low-power CPUs.

Traditionally, the main showstopper for RDMA was that it requires special high-end hardware (NICs and possibly switches). However, recently several RDMA stacks implemented entirely in software have been proposed, such as SoftiWARP [15] or SoftRoCE [3]. We use the term *soft-RDMA* to refer to any RDMA stack implemented in software. While soft-RDMA cannot provide the same performance as hardware-based solutions, it has the advantage that it runs on top of standard Ethernet and requires no additional hardware support. The modifications we later propose for Memcached will be leveraging the one-sided operations of RDMA, namely *read* and *write*. Let us quickly revisit how those operations work using a soft-RDMA implementation (see Figure 4).

Soft-RDMA is typically implemented inside the kernel as a loadable kernel module. Assuming a client wants to read some data from a remote server's main memory, it will first have to allocate a receiving buffer on its own local memory. The client then registers the address and length of the receiving buffer with the in-kernel soft-RDMA provider. Similarly, the server will have to register the memory to be accessed by the client with its own local soft-RDMA provider. Registering memory with RDMA causes the RDMA provider to pin the memory and return an identifier (stag, typically 4 bytes) which can later be used by the client to perform operations on

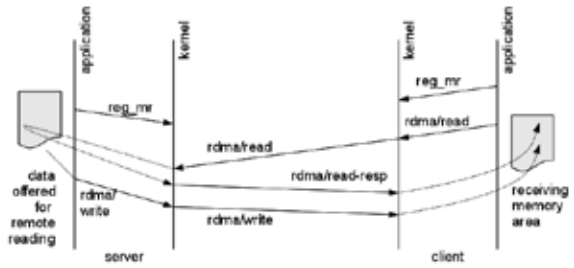


Figure 4: Example of a one-sided RDMA read operation

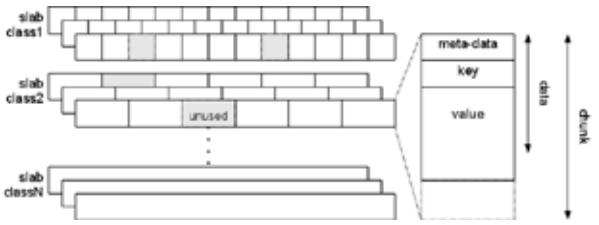


Figure 6: Memory management in Memcached

that memory. The actual data transfer is triggered by the client who issues a RDMA read operation containing the stag associated with the memory of the remote server.

Depending on which soft-RDMA stack is used, the RDMA read request may be transmitted to the server using in-kernel TCP or some proprietary transport protocol. At the server side, the read request is not passed on to the application but served directly by the in-kernel soft-RDMA stack, which – given the stag specified by the client – directly triggers transmission of the requested memory without any extra copying of the data. Copying is not avoided at the client side, but the RDMA receive path is typically shorter than the socket receive path.

In addition to read, soft-RDMA also supports write operations in a similar manner. Those operations are called one-sided since they only actively involve the client application without having to schedule any user level process at the server side. Besides avoiding the scheduling of the server process, another advantage of RDMA read/write operations is their zero copy transmit feature which yields a lower CPU consumption compared to the traditional socket based approach.

3 Memcached/RDMA

Memcached employs a multi-threaded architecture where a number of worker threads access the single hash table (Figure 5a). Serialized access to individual keys is enforced with locks. Requests entering the system will be demultiplexed to the various threads by using a dedicated dispatcher thread that monitors the server socket for new events. A key building block of Memcached is the memory management, implemented to avoid memory fragmentation and to provide fast re-use of memory blocks (see Figure 6). Memory is allocated (using malloc) in slabs of 1 MB size. A slab is assigned to a certain slab class which defines the chunk size the slab is broken into, e.g., slab classes may exists for chunk sizes of 64, 128 and 256 bytes, doubling all the way up to 1 MB. Each slab maintains a list of free chunks, and when a request (e.g., SET) comes in with a particular size, it is rounded up to the closest size class and a free chunk is taken from one of the slabs in the class. Items stored inside chunks contain key and value as well as some meta data. Chunks that are not used are put back into the free list to be re-used at a later point in time.

3.1 Proposed Modifications

The architecture of Memcached does contain several potential inefficiencies that can result in a significant amount of load for low-frequency CPUs. First, data is copied between the kernel and the worker threads. Given that the operations of Memcached (SET/GET) are rather simple and do not require much processing, the overhead introduced by copying data from user space to kernel and vice versa becomes significant (as shown in Figure 3). Second, in most cases processing a Memcached client request requires a context switch as the corresponding worker thread needs to be scheduled. And third, the time required to parse client requests – although small when measured in absolute numbers – can be a significant fraction of the overall processing time of a single request.

All three of those issues are addressed in our RDMA-based Memcached approach which takes advantage of RDMA's one-sided operations. In particular, we propose modifications of Memcached in the following three aspects:

1. **Memory management:** We associate memory chunks of Memcached with registered memory regions in soft-RDMA
2. **GET operation:** We use RDMA/read to implement the Memcached/GET operation.
3. **Parsing:** We increase the level of parsing necessary at the client side in favor of lowering parsing the effort at the server.

To simplify the upcoming description of the system, we henceforth use the name Memcached/RDMA when referring to the modified Memcached, and simply Memcached when referring to the unmodified system. The high-level architecture of Memcached/RDMA is shown

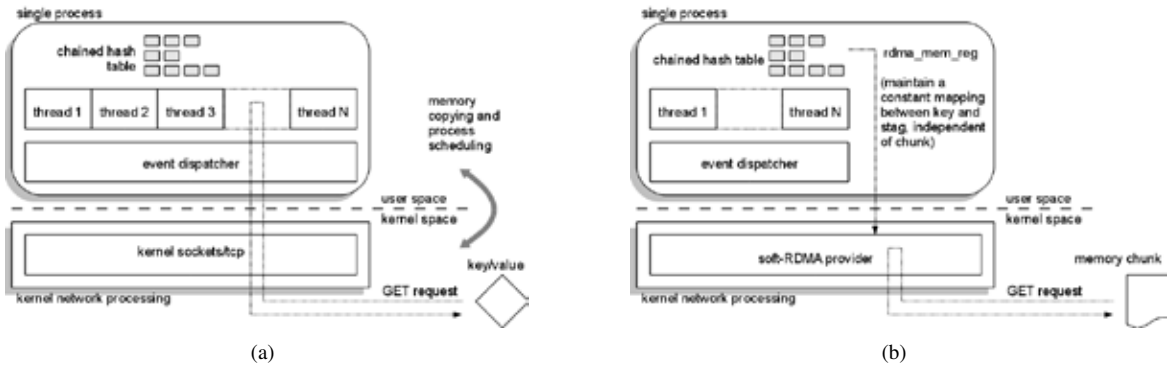


Figure 5: Architecture of (a) Memcached and (b) Memcached/RDMA

in Figure 5b. For a detailed understanding of the proposed modifications, let us walk through some aspects of the memory management and through the two main operations of Memcached, SET and GET:

Memory management: In Memcached, memory chunks of fixed sizes are the basic unit for storing key/value pairs. The main idea in Memcached/RDMA is to allow chunks to be read remotely without involving the Memcached user-level process. For this, Memcached/RDMA registers every newly allocated chunk with the soft-RDMA provider. The RDMA stag returned by the memory registration call is stored inside the meta data of the chunk.

SET: A client of Memcached/RDMA issues a SET request using TCP, just as if it was connected to a unmodified Memcached server. Upon receiving the SET request, the Memcached/RDMA server finds a chunk that fits the item (key/value pair) size, stores the item inside that chunk and inserts the chunk into the global hash table. If the item is a new item (the key does not yet exist in the hash table), the SET operation transmits, as part of the response message, the stag stored inside the chosen chunk. If the item replaces an already existing item with the same key, then Memcached/RDMA swaps the stags stored in those two chunks, updates the RDMA registration for both chunks, and also includes the stag of the newly created item in the response message. Clients maintain a table (stag table) matching keys with stags for a later use. *It is important to note that the swapping of stags guarantees that the same stag is used throughout the lifetime of a key, even as the chunk storing the actual key/value pair changes.*

GET: Before issuing a GET operation, the client checks whether an entry for the given key exists in the stag table. Depending on whether a stag is found or not, the GET operation is executed in the following ways:

If no entry is found, the GET operation is initiated using the TCP-based protocol. The server, however, rather than responding on the reverse channel of the TCP connection, will use a one-sided RDMA/write operation to transmit the requested key/value pair to the client using additional RDMA information that was inserted into the client request. Besides the key/value pair, the respond message from the server also includes the stag associated with the chunk storing the item at the server; this stag is inserted into the local stag table by the client.

If a stag entry is found at the client before the GET request is issued, Memcached/RDMA will use a one-sided RDMA/read operation to directly read from the server the chunk storing the requested key/value pair. The chunk, once received by the client, is parsed and the value contained in the chunk is returned to the application. The client also verifies that the key stored inside the received chunk matches the key which was requested. In case of an error the client purges the corresponding entry in the stag table and falls back to the first mode of operation.

Leveraging one-sided semantics during the GET operation has several advantages: First, data sent back to the client is always transmitted without extra copying at the server, thereby saving CPU cycles. This is true for both modes of the GET operation, namely when using RDMA/write as well as when using RDMA/read. Second, the Memcached/RDMA user level process is only involved in SET requests and in first-time GET requests by clients. All subsequent GET requests are handled exclusively by the operating system's network stack, lowering the load at the server. And third, directly reading entire chunks from the server using RDMA/read eliminates the need for parsing client requests, decreasing the CPU load at the server further.

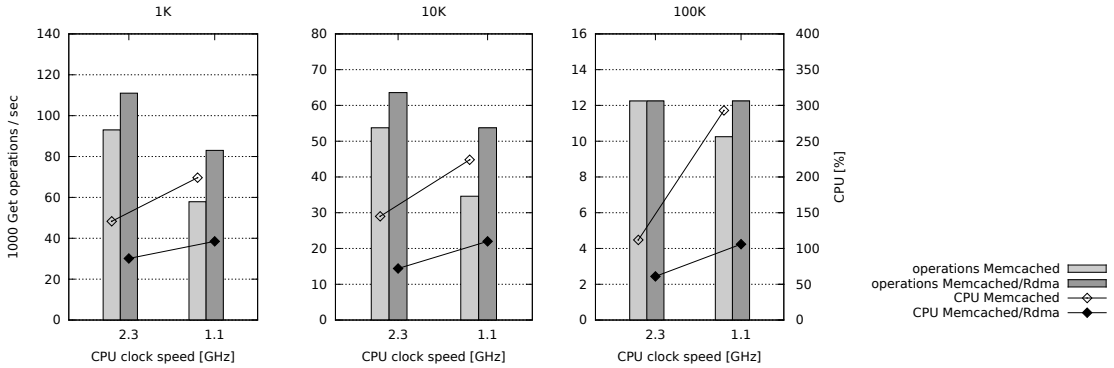


Figure 7: Performance and CPU consumption of Memcached and Memcached/RDMA for different value sizes (1K, 10K, 100K) and different server CPU clock frequencies (2.3 GHz, 1.1 GHz)

3.2 Lazy Memory Pinning

The proposed modifications, if implemented as described, can lead to a waste of memory resources if only a small fraction of the allocated chunks in Memcached/RDMA is actually read and written by the clients. As mentioned in Section 2, registering memory buffers with RDMA would typically cause the associated memory to be pinned, requiring the commitment of real physical memory. It is recommended to run Memcached with sufficient physical memory to avoid paging and the corresponding performance degradations, but overcommitting the memory resources is technically possible and supported. To maintain this feature in Memcached/RDMA, we employ a novel lazy memory pinning strategy. Any memory registered with soft-RDMA will not be pinned until it is accessed for the first time through read or write operations. Outside of RDMA operations, memory can be paged out, causing extra overhead to page in the virtual memory at the subsequent RDMA operation. If sufficient memory is available to keep all the chunks pinned, no overhead would occur. If, however, the underlying system is short of physical memory then the memory accessed by RDMA will be paged in and out according to the same OS principles any user-level memory is managed, e.g., hot memory typically stays in main memory whereas cold memory gets swapped to disk. However, we did not evaluate this feature in our experiments.

4 Evaluation

Memcached already supports both UDP and TCP transports. Thus, one appealing approach to implement Memcached/RDMA would be to just add an RDMA transport to Memcached. In this work, however, we have implemented a standalone prototype of Memcached/RDMA by re-using some of Memcached's original data structures. As one example of a soft-RDMA provider we use SoftiWARP [15], an efficient Linux-based in-kernel RDMA stack. Memcached/RDMA does not interact with SoftiWARP directly, but builds on libibverbs, which is a standardized, provider-independent RDMA API. Separating Memcached/RDMA from the actual RDMA provider will make it possible in the near future to also exploit hardware accelerated RDMA implementations (e.g., Infiniband).

Configuration: Experiments are executed on a cluster of 7 nodes, each equipped with a 4 core Intel Xeon E5345 CPU and a 10 GbE adapter. One of the nodes in the cluster acts as server running Memcached with 8 threads, whereas the other nodes are used as clients. Depending on the actual experiment, the server is configured in low-power mode, which causes the CPU to run at 1.1 GHz clock frequency. Experiments consist of two phases: in the first phase, clients insert a set of 1000 key/value pairs into Memcached; in the second phase, the clients query Memcached using GET calls at the highest possible rate. We used OProfile [2] to measure the CPU load at the server.

Performance: Figures 7 shows a performance comparison of Memcached and Memcached/RDMA in terms of number of operations executed per second at the server. Each panel illustrates an experiment for a given size of key/value pairs (1K, 10K, 100K); performance numbers are given for both 2.3 GHz and 1.1 GHz CPU clock speed. Note that the range marked by the y-axis (GET operations) differs for each panel. It is easy to see that Memcached/RDMA outperforms unmodified Memcached in almost all cases by 20% or more. The performance gap is visible for 2.3 GHz CPU clock speed and

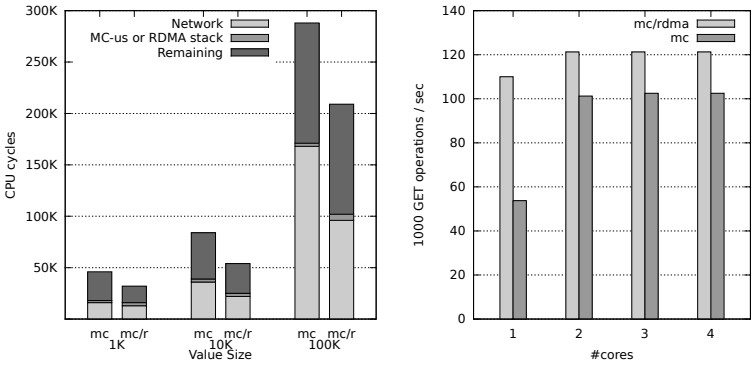


Figure 8: (a) CPU efficiency ("mc" for Memcached, "mc/r" for Memcached/RDMA, "MC-us" for Memcached user space) and (b) performance for different numbers of cores.

increases even further once the server switches to low-power mode. An exception is the configration with 100K size key/value pairs and 2.3GHz clock speed where both Memcached and Memcached/RDMA perform equally well. The explanation for these performance results lies in the CPU consumption of Memcached and Memcached/RDMA, which we plot against the right y-axis (with 400% referring to all four cores being fully loaded) in each panel. While four cores running at 2.3GHz can handle bandwidth intensive 100K size GET requests, the difference in CPU load between Memcached and Memcached/RDMA eventually turns into a performance advantages for Memcached/RDMA at 1.1GHz clock speed.

To understand how exactly the CPU efficiency of Memcached/RDMA is materialized, we again divided the used CPU cycles into three classes "network", "application" and "remaining" (see Section 1 for descriptions of these classes). Here, the class "application" refers to the actual RDMA processing in the kernel (label "RDMA stack" in Figure 8a) since the user space Memcached process is entirely outside of the loop during GET operations. Figure 8a illustrates the used cycles per GET operation of Memcached/RDMA and compares them to the numbers we have previously shown for the unmodified Memcached (see Table 2). Clearly, Memcached/RDMA is more CPU efficient for all three sizes of key/value pairs. For smaller key/value sizes the efficiency stems partially from a low operating system involvment (e.g., context switching). For larger key/value sizes the efficiency is mostly due to zero-copy networking.

Efficiency: One obvious opportunity to increase the performance of Memcached is to throw more cores at it. This approach – besides being limited by the current scalability walls of Memcached [6, 9] – reduces the CPU efficiency of Memcached. Figure 8b illustrates that Memached/RDMA when using just a single core is able to provide a similar performance as the unmodified Memached with all four cores.

The key observation from these experiments is that Memcached/RDMA uses CPU cycles more efficiently which allows for either handling more operations per second, or for using fewer cores than a comparable unmodified Memcached server.

5 Related Work

There exists obviously a substantial body of work on key/value stores, a large part thereof adopting a disk-based storage model focusing mostly on scalability [8, 7, 14]. Fawn [5] is a key/value store designed for Flash and low-power CPUs. Like other key/value stores targetting low power [6], Fawn does not consider network speeds greater than 1GbE. Using RDMA to boost distributed storage and key/value stores has been proposed in [12, 10], and partially in [13]. These works, similar to our work, do successfully use RDMA to improve the performance of distributed systems. However, comparing the results with our work is difficult as those systems rely on dedicated RDMA hardware which is often not available in commodity datacenters. In addition, the main focus of these works is on improving throughput and latency, and not so much on reducing the CPU footprint of the system.

In contrast, our work explicitly studies high-speed networks in a low-power CPU configuration. The solution we propose leverages one-sided operations in RDMA which improves both the performance as well as the CPU consumption of the system. Furthermore, our approach can be applied within commodity data centers as it is

purely implemented in software without requiring dedicated hardware support.

6 Conclusion

In this paper, we studied the feasibility of implementing an in-memory key/value store such as Memcached on a cluster of low-power CPUs interconnected with 10 Gigabit Ethernet. Our experiments revealed certain inefficiencies of Memcached when dealing with high link speeds at low CPU clock frequencies. We proposed modifications to the Memcached architecture by leveraging one-side operations in soft-RDMA. Our approach improves the CPU efficiency of Memcached due to zero-copy packet transmission, less context switching and reduced server-side request parsing. As a result, we were able to improve the GET performance of Memcached by 20%. While this paper looked at in-memory key/value stores, we believe that many of its ideas can be extended to key/value stores targeting non-volatile memory (e.g., Flash, PCM).

Acknowledgement

We thank the anonymous reviewers and our shepherd, John Carter, for their helpful comments and suggestions.

References

[1] Memcached - a distributed memory object caching system. http://memcached.org.

[2] OProfile - An Operating System Level Profiler for Linux. http://oprofile.sourceforge.net.

[3] Softroce. www.systemfabricworks.com/downloads/roce.

[4] ADIGA, N., ET AL. An overview of the bluegene/l supercomputer. In *Supercomputing, ACM/IEEE 2002 Conference* (nov. 2002), p. 60.

[5] ANDERSEN, D. G., FRANKLIN, J., KAMINSKY, M., PHANISHAYEE, A., TAN, L., AND VASUDEVAN, V. Fawn: a fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 1–14.

[6] BEREZECKI, M., FRACHTENBERG, E., PALECZNY, M., AND STEELE, K. Many-core key-value store. In *Green Computing Conference and Workshops (IGCC), 2011 International* (july 2011), pp. 1 –8.

[7] COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow. 1*, 2 (Aug. 2008), 1277–1288.

[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 *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 205–220.

[9] GUNTHER N., SUBRAMANYAM S., PARVU S. "Hidden Scalability Gotaches in Memcached" (June 2010).

[10] JOSE, J., SUBRAMONI, H., LUO, M., ZHANG, M., HUANG, J., WASI-UR RAHMAN, M., ISLAM, N. S., OUYANG, X., WANG, H., SUR, S., AND PANDA, D. K. Memcached design on high performance rdma capable interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing* (Washington, DC, USA, 2011), ICPP '11, IEEE Computer Society, pp. 743–752.

[11] LANG, W., PATEL, J. M., AND SHANKAR, S. Wimpy node clusters: what about non-wimpy workloads? In *Proceedings of the Sixth International Workshop on Data Management on New Hardware* (New York, NY, USA, 2010), DaMoN '10, ACM, pp. 47–55.

[12] MAGOUTIS, K., ADDETIA, S., FEDOROVA, A., SELTZER, M. I., CHASE, J. S., GALLATIN, A. J., KISLEY, R., WICKREMESINGHE, R., AND GABBER, E. Structure and performance of the direct access file system. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2002), ATEC '02, USENIX Association, pp. 1–14.

[13] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J., AND ROSENBLUM, M. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 29–41.

[14] POWER, R., AND LI, J. Piccolo: building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–14.

[15] TRIVEDI, A., METZLER, B., AND STUEDI, P. A case for rdma in clouds: turning supercomputer networking into commodity. In *Proceedings of the Second Asia-Pacific Workshop on Systems* (New York, NY, USA, 2011), APSys '11, ACM, pp. 17:1–17:5.

Revisiting Software Zero-Copy for Web-caching Applications with Twin Memory Allocation*

Xiang Song† ‡, Jicheng Shi† ‡, Haibo Chen†, Binyu Zang† ‡
†Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
‡Software School, Fudan University

ABSTRACT

A key concern with zero copy is that the data to be sent out might be mutated by applications. In this paper, focusing specially on web-caching application, we observe that in most cases the data to be sent out is not supposed to be mutated by applications, while the metadata around it does get mutated. Based on this observation, we propose a lightweight software zero-copy mechanism that uses a twin memory allocator to allocate spaces for zero-copying data, and ensures such data is unchanged before being sent out with a lightweight data protection mechanism. The only change required to an application is to allocate zero-copying data through a specific ZCopy memory allocator. To demonstrate the effectiveness of ZCopy, we have designed and implemented a prototype based on Linux and ported two applications with very little effort. Experiments with Memcached and Varnish shows that show that ZCopy can achieve up to 41% performance improvement over the vanilla Linux with less CPU consumption.

1 INTRODUCTION

Many network-intensive applications can easily be limited by the speed of network I/O processing. Other than the physical limitation of networking devices, the performance of networking applications are also constrained by the efficiency of network I/O sub-systems, in which data copying is one of the key limiting factors. Usually, during network protocol processing, the operating system kernel has to copy data from user space to a kernel buffer and then sends the kernel buffer to the network device.

Though there has been extensive research on avoiding the data copying, prior systems are still not easily adoptable for many applications on commodity operating systems with commodity networking devices. One approach is bypassing the operating system with Remote DMA. However, these require special and expensive hardware (e.g., Infiniband [2] and Myrinet [10]) and most commodity networking devices have not been built with such support. Several previous software zero copy mechanisms, such as fbufs [7] and IO-Lite [11] are designed for a micro-kernel and require special data management and accessing methods across protection domains. Container shipping [12] supports zero-copy on UNIX platforms, but requires data being aggregated in a scatter-gather manner and additional system-call interfaces. Approaches [5, 6] using on-demand memory mapping and copy-on-write mechanism are limited by the protection granularity (e.g., page size) and the corresponding alignment requirement, thus may face the false sharing problem that protects unwanted data. This may cause notable performance overhead for irregular (e.g., unaligned) data chunks. Modern operating systems also have several mechanisms to support zero copy, such as sendfile [14] and splice [9]. However, such mechanisms require zero-copying data to be treated as files, which is not feasible in many applications that need to mutate the data to be sent out.

The key issue in supporting software zero-copy is that the zero-copying data might be mutated when being sent out. This is because when a user application invokes a data sending system call (e.g., sendmsg and write), it assumes that the data has been sent out when the system call returns. However, when such system calls return, the data might have not been moved into the networking devices. If the kernel does not copy the data from the user buffer to a kernel buffer, any changes on the data from applications may be sent out, which violates the semantics of such system calls.

Intuitively it should be the case that the data will normally not be mutated. However, focusing specifically on web-caching applications, we observe that, in most cases, the data to be sent out is not supposed to be mutated by applications. However, the data around it, especially the metadata corresponding to it, does get mutated. Some metadata (e.g., the data expire time in Memcached [8]) is usually co-located around the data to be sent. Due to lacking of application semantics, operating

*We thank our shepherd Alexandra Fedorova the anonymous reviewers for their insightful comments. This work was funded by China National Natural Science Foundation under grant numbered 61003002, a grant from the Science and Technology Commission of Shanghai Municipality numbered 10511500100. Xiang Song was also funded by Fudan University's outstanding doctoral research funding schemes 2011.

systems cannot simply zero-copy a page with specific network data packets as that page holding the network data can be modified by applications.

Based on the above observation, we revisit the software zero-copy mechanism for web-caching applications. The basic idea is using a second (twin) memory allocator to allocate and aggregate data that are likely to be zero-copied, and providing a lightweight memory protection mechanism in case such data does get modified. Hence, the zero-copying data can be isolated from other application data, thus can be aggregated together to allow kernel to use traditional page-level protection. This minimizes unnecessary write protection faults due to false sharing. To support software zero copy, an in-kernel proxy is added into the UDP and TCP processing paths to distinguish the zero-copy data with the others. A write protection module is also added to handle rare cases where the data that is supposed to be zero-copied have really been mutated. In such a case, the data will be copied to ensure program correctness.

We have implemented a prototype based on Linux 2.6.38. The prototype of ZCopy is very lightweight and adds around 735 lines of code (LOCs) to Linux kernel and adds 20 LOCs to streamflow [13]. It consists of a specific user-level memory allocator ZC_alloc based on streamflow. A 200 LOCs user-level library is implemented to support cooperation between the ZCopy kernel and the ZC_alloc to provide memory protection for zero-copying data.

The porting effort required to run web-caching applications on ZCopy using zero-copy mechanism is also quite small. Providing zero-copy support to Memcached [8], a widely-used key-value based memory caching server, requires only 10 LOCs changes. Running Varnish [4] server also only requires 3 LOCs modification. The only change required is simply replacing the memory allocator for zero-copying data with the one provided by the ZC_alloc.

To measure the effectiveness of ZCopy, we conducted several application performance measurements using Memcached and Varnish web caching system. Performance results show that ZCopy brings modest improvement over vanilla Linux. ZCopy improves the throughput of Memcached over vanilla Linux up to 41.1% and 40.8% for UDP and TCP processing when the value size is larger than 256 bytes. The performance speedup of Varnish ranges from 0.7% to 7.9% for data size ranging from 2 KBytes to 8 KBytes.

2 Observation

To gain insight into how network data might be mutated, we make a case study on Memcached. Figure 1 shows the basic storing item structure of Memcached to store key/value pairs. The key/value data is stored at the end of *stritem*, while the metadata is stored from the beginning of it. Each time Memcached receives a request and find a corresponding key/value pair, the refcount of the corresponding item will be increased in function *do_item_get* (As shown in Figure 2). If we write protect the key/value pair, we need also write protect the metadata around it. Hence, there will be a lot of unnecessary protection faults due to false sharing.

The example indicates that for some networking applications, the network I/O data to be sent out is not supposed to be mutated. However, the data around it, especially the metadata corresponding to it does get mutated. Hence, naively write-protecting the networking data may also protect the metadata allocated within the same page, resulting in false protection.

```
typedef struct _stritem {
1   struct _stritem *next;
2   ...
3   uint8_t          nkey;     /* key length */
4   void * end[];              /* struct {  keyn
                                            suffix
                                            data  } */
} item;
```

Figure 1: *Memcached storing item structure.*

```
1 item *do_item_get(const char *key, const size_t nkey) {
2       item *it = assoc_find(key, nkey);
3       ...
4       if (it != NULL) {
5               it->refcount++;
6       ...
7 }
```

Figure 2: *Code piece of function do_item_get.*

3 Design and Approaches

This section first presents an overview of ZCopy and then illustrates the approaches to supporting efficient zero-copy mechanism.

3.1 ZCopy Overview

It is quite intuitive to let applications to designate which data should be zero-copied. When such data is being sent out, ZCopy will zero-copy it while processing other data through the normal path. However, it has to deal with the following issues: 1) it should retain the existing memory accessing manner for user applications; 2) it should conform to existing system calls to avoid adding any new interfaces; and 3) it should provide proper protection over the data to be sent out to conform to the semantics of existing network sending system calls.

In ZCopy, we introduce a twin memory allocator to separately allocate data according to application semantics and aggregate several zero-copying memory blocks into the same memory chunks. Hence, the data to be

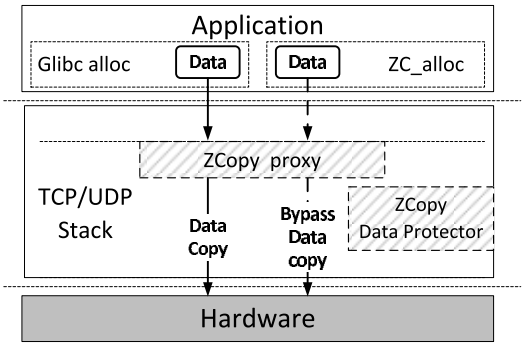


Figure 3: *An overview of architecture of ZCopy*

protected can be separated from other application data. Figure 3 shows the general architecture of ZCopy. The application running on ZCopy can use the original memory allocator (e.g., glibc) to allocate memory for normal data or use the twin memory allocator named ZC_alloc to allocate memory for zero-copying data. A ZCopy proxy is added to the UDP and the TCP package processing path to distinguish the network data that will be zero-copied from others. If the data is allocated from ZC_alloc, ZCopy will bypass the data copy path. Otherwise, ZCopy will handle the data as usual. The proxy also cooperates with the ZCopy data protection module to provide basic write protection on the zero-copying data.

3.2 Supporting Zero-copy

3.2.1 Isolating Zero-copying Data with Twin Memory Allocator

To isolating zero-copying data from other data, ZCopy provides a twin memory allocator along with the original one to allocate memory for network data that is guaranteed to be insulated from other data allocated from a generic memory allocator (e.g., glibc). Restricted by the minimal memory protection granularity of a page size and the following address alignment requirement, ZC_alloc has to pay special attention to small memory blocks (e.g., block size small than 1024 bytes). A naive way to handle this is to allocate one page for each request. However, this may waste a lot of memory. ZC_alloc uses an aggressive way by aggregating memory blocks with similar sizes into the same basic memory unit, namely the pageblock. A pageblock is treated as a basic protection chunk and usually consists of several pages (16 pages by default). It is write protected only when it is full of zero-copying data. As ZC_alloc aggregates zero-copying data together to provide memory protection, it minimizes the amount of wasted memory (e.g., by aggregating small objects smaller than 1 page size into a default pageblock, the maximum amount of memory wasted is less than 1 page, which is less than 6.25%).

If the allocation request is for a large data block, ZC_alloc directly allocates a memory chunk rounded from the requesting size. A threshold (4096 bytes by default) is set in ZC_alloc to decide whether a request is for the large data block. This threshold can be tuned by the programmer if needed.

The twin memory allocator is especially friendly to the reusable data. Once a data block is allocated it will be sent out to network multiple times before it is modified or freed. One representative usage scenario is allocating value data for Memcached. Memcached server caches a lot of key/value pairs in memory to serve quick key/value queries. Every time the server receives a request containing a key, it will respond with the value corresponding to that key. For the perspective of long execution, the key/value pairs are not expected to be modified or freed. Hence, we can zero-copy the value during data transferring without worrying about the modification to such data in most cases.

3.2.2 Zero-copying Network I/O Data

ZCopy supports two common network protocols: UDP and TCP. We add a proxy in UDP and TCP's package processing paths to distinguish the network data that will be zero-copied and others. At the very beginning, ZCopy will first check whether current process wants to use zero-copy mechanism or not. If so, it will check whether there are any memory blocks that need to be write protected. The ZCopy data protection module is invoked if write protection is needed.

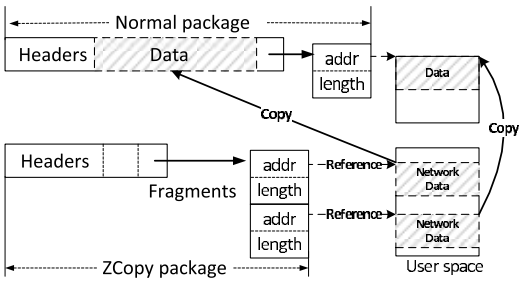


Figure 4: *The structure of normal package and ZCopy package*

ZCopy handles zero-copying data at the time when the network data is organized into a network package. Figure 4 shows the structure of normal network package and the ZCopy package. In normal cases (shown in the top half of Figure 4), a network package consists of several protocol headers followed by network data. The network data can be organized as a single data buffer or a list of data buffers. The data is copied from user address space into the package in order. If the package buffer is not large enough to hold all network data, the kernel will allocate new empty pages to hold the rest of the data and attaches them into the package's page fragment list. Each entry in the list contains the starting address of the data

and its length. When the package is passed to the NIC driver, the driver will first transfer the package content and the fragments to the NIC hardware through the DMA engine.

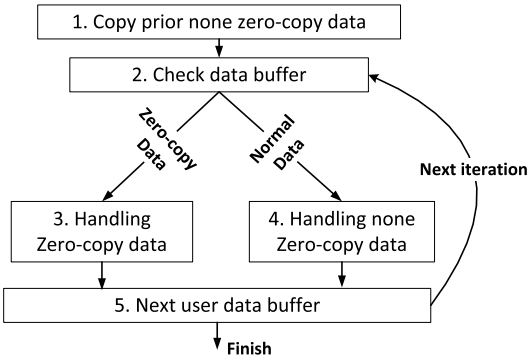


Figure 5: *Zero-copy in UDP package processing*

ZCopy treats zero-copying data differently from normal data. Each pageblock is identified by a magic string, thus the zero-copy data buffers can be distinguished with others. We use the UDP package processing as an example to illustrate the process of handling zero-copying data. Figure 5 shows the UDP package processing path in ZCopy and the bottom half of Figure 4 shows the structure of a ZCopy package. ZCopy first scans the user data buffer lists to copy all prior normal data into the network package buffer including the protocol headers (step 1). Then, it iteratively processes the following user buffers by handling zero-copying data and normal data separately (step 2-5). It will check the pageblock magic string to discover zero-copying user buffers. For zero-copying data (step 3), it first gets the starting address and the length of the data buffer. It then finds all pages covered by the data buffer and finally organizes the pages in the form of fragments and adds them into the package's page fragment list. For normal data (step 4), it allocates new empty pages and copies the buffer content into them. ZCopy finally organizes the pages in the form of fragments and adds them into the package's page fragment list. The package will be passed into the lower level of the network stack.

One optimization to the ZCopy proxy is to treat read-only data buffers as zero-copying buffers, though they are not allocated using ZC_alloc. This can simply be done by feeding the offset and length in the fragment list.

3.2.3 Protection of Zero-copying Data

ZCopy must provide a protection mechanism to the zero-copying data in case it is mutated when the data is sent out. To do this, ZCopy adds a simple data protection module into the native memory management system.

Based on the page-level protection granularity in kernel, small data blocks allocated from ZC_alloc are batched in a group and are treated as a whole for write protection. The minimal protection unit is one pageblock. When a pageblock is full, ZC_alloc will request the kernel to protect it. To avoid the cost of context switches between user space and kernel space and possible false protection problem caused by early write protection, ZCopy batches the requests from ZC_alloc to delay the protection of the pageblock until the system enters the network package processing path. The protection is done by walking the page table of the target range and changing the protection bit of the corresponding page table entries. When the pageblock is not full, data allocated from ZC_alloc are still sent through normal path without being zero-copied.

ZCopy tries to protect zero-copying data blocks in an aggressive way. ZCopy does not remove the write protection of the data block even if the data block is completely sent out by the hardware. The removal of the write protection is triggered only when a write operation is trapped by the kernel. At that time, the reference count of the page corresponding to the faulting address is first checked. If the count is larger than one, a copy-on-write mechanism is used to protect the network data from being modified. Otherwise, the changing request should come from the application itself and we simply remove the write protection. Note that, the basic protection unit is a pageblock, any write to a write protected pageblock will cause all the data blocks belong to the pageblock lose the write protection. However, we do not expect this happens frequently as mutation on zero-copying data is rare.

4 Experimental Results

All experiments were conducted on an Intel machine with 2 1.87 Ghz Six-Core Intel Xeon E7 chips running Debian GNU/Linux 6.0 with the kernel version 2.6.38. The NIC used is an Intel 82576 Gigabit Network Controller. We use another Intel machine with the same hardware and software configuration as the client machine. To minimize the interaction between different cores of a multi-core system (e.g., cache trashing), experiments were conducted using only one CPU core.

We use two widely-used web-caching applications, Memcached 1.4.5 [8] and Varnish 3.0.0 [4] to demonstrate the performance improvements. All applications in the experiments use the ZC_alloc to allocate memory for network data to eliminate the effect of using different memory allocators.

4.1 Memcached

Memcached [8] caches multiple key/value pairs in memory. Each time it receives a request containing a key, it will respond with the corresponding value. From a long run's perspective, the key/value pairs are not expected to

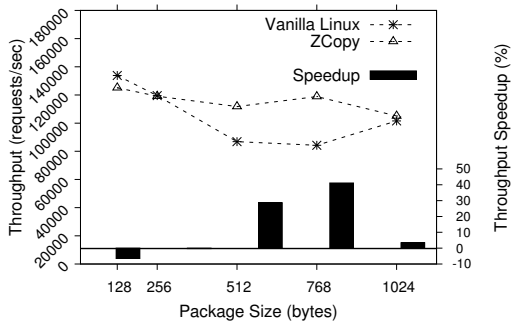


Figure 6: *The throughput of Memcached in ZCopy and vanilla Linux with UDP and the speedup of ZCopy.*

be modified or freed. However, the metadata (e.g., the data expire time, the item links) along with the cached pairs may change. We modify Memcached to allocate memory for the values from ZC_alloc. This takes only 10 lines of modification to the original Memcached.

We use the memaslap testsuite form the libmemcached library [3] as the client of Memcached. The client first warms up Memcached with a user-defined number of key/value pairs and then randomly issues get and set operations through several concurrent connections.

UDP: Figure 6 shows the average throughput of Memcached in ZCopy and vanilla Linux. The Memcached is warmed up with ten thousand key/value pairs. The memaslap client is configured to issue pure get operations through 36 concurrent connections from 12 threads using the UDP protocol. We adjust the number of worker threads of Memcached to achieve the best performance. The CPU usage in all cases is above 99%. Vanilla Linux performs slightly better when the value size is smaller than 256 bytes. However, when the value size reaches 512 bytes, ZCopy starts to outperform vanilla Linux. In 512 bytes cases, ZCopy has a 28.7% performance improvement. When the value size is 768 bytes, the performance improvement increases to 41.1%. For the case where the value size is 1024 bytes, ZCopy and vanilla Linux has nearly the same throughput as the network reaches its hardware limitation.

The performance improvement comes from two parts: 1) minimized data copying and 2) reduced cache trashing. Figure 7 compares the time spent on UDP package processing in ZCopy and vanilla Linux. In ZCopy, the package processing time is around 3000 cycles in all cases. However, in vanilla Linux, the time increases along with the package size and reaches 4400 cycles in 1024 bytes cases. Table 1 shows the L2 cache miss rate of Memcached in Linux and ZCopy. ZCopy reduces more than 10% L2 cache misses in UDP cases. The hottest function *copy_user_generic_string* in Linux disappears in ZCopy. Another reason for such notable performance improvement in the 512 and 768 cases is that the

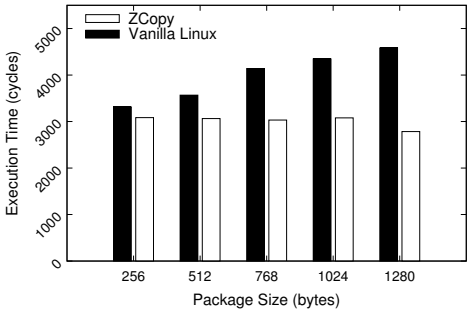


Figure 7: *The time spent on UDP package processing for Memcached in ZCopy and vanilla Linux.*

shorter package sending time in ZCopy causes the NIC interrupt handler switch frequently to the polling mode which is more effective than the interrupt mode in heavy network stress. However, in vanilla Linux, the network status triggers less frequent switches to the NIC polling mode.

L2 Cache Miss Rate (1 miss/K cycles)			
	512 bytes	768 bytes	1024 bytes
UDP Linux	4.89	5.17	6.11
UDP ZCopy	4.17	4.57	4.73
TCP Linux	8.08	9.06	10.86
TCP ZCopy	7.73	8.22	9.46

Table 1: *The L2 cache miss rate in vanilla Linux and ZCopy in 256 byte, 768 byte and 1024 byte cases.*

TCP: Figure 8 shows the average throughput of Memcached in ZCopy and vanilla Linux and the performance speed of ZCopy over vanilla Linux. We use the same evaluation method used in the UDP experiments. For each TCP connection, we only issues a signle request and then close it. Vanilla Linux performs better when the value size is smaller than 256 bytes. However, when the value size reaches 512 bytes, ZCopy starts to outperform the vanilla Linux by 40.8%. When the value size is with 1024 bytes, ZCopy outperforms vanilla Linux by 30.8%. The performance of Memcached reaches the hardware limits when the value size is of 2048 bytes.

As in UDP, the performance improvement comes from copy avoidance and reduced cache trashing. As the code for TCP package processing and data sending is mixed together, we measure the time spent on the *tcp_sendmsg* instead of TCP package processing time. Figure 9 shows the profiling results. From the figure we can see that ZCopy does reduce the time spent on *tcp_sendmsg* in all cases. Table 1 shows the L2 cache miss rate of Memcached in Linux and ZCopy. ZCopy reduces 10.2% L2 cache misses in 768 byte cases and 14.8% L2 cache misses in 1024 byte cases.

4.2 Varnish

Varnish [4] is an open-source web application accelerator. It caches web content into memory objects and re-

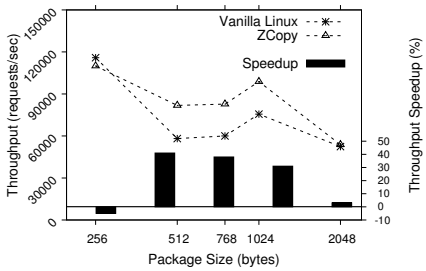


Figure 8: *The throughput of Memcached in ZCopy and vanilla Linux with TCP and the speedup of ZCopy.*

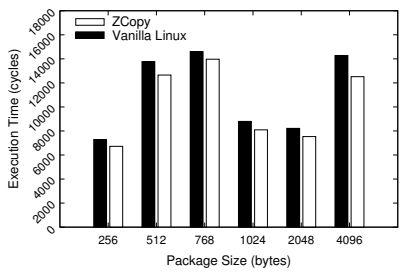


Figure 9: *The time spent on function tcp_sendmsg for Memcached in ZCopy and vanilla Linux.*

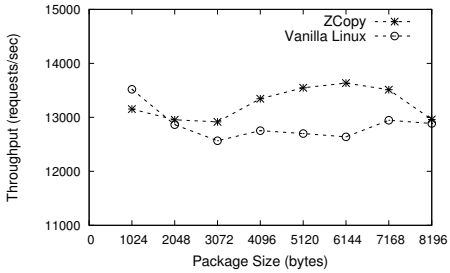


Figure 10: *The throughput of varnish server in ZCopy and vanilla Linux.*

turns web objects according to the network request. We modify Varnish to allocate object memory from ZC_alloc with 3 LOCs changes.

We test Varnish using ab (apache benchmark) from Apache with the web page sizes ranging from 1 KBytes to 8 KBytes (the average individual response size ranges from 3 KBytes to 15 KBytes [1].) Figure 10 compares the performance of ZCopy and vanilla Linux. The varnish server saturates the CPU on both ZCopy and vanilla Linux. Vanilla Linux performs slightly better with small web page sizes (1 KBytes). However, when the web page size increases, ZCopy starts to outperform Linux. The performance improvement reaches 7.8% when the web page size increases to 6 KBytes. Both configurations reach networking limitation when the web page size increases to 8 KBytes. The reason that the improvement is much less than Memcached is that the single request processing time in Varnish is much longer than that in Memcached, which thus amortize the improvements of ZCopy.

	CPU cycles
getpid	1149.9
ZCopy write protection fault	2802.5
native page fault	6247.4

Table 2: *The execution time of invoking getpid system call, triggering ZCopy write protection fault and triggering native page fault.*

4.3 ZCopy Primitive

Overhead of Write Protection We also evaluate the cost of triggering write protection faults for zero-copied data. Table 2 shows the execution time of invoking the getpid system call, triggering ZCopy write protection fault and triggering traditional page fault respectively. The cost of triggering a ZCopy write protection fault is much smaller than triggering a native page fault. This is because usually ZCopy only removes the write protection of the faulting address from the page table, which is much less expensive.

5 Conclusion and Future Work

This paper revisited the existing software zero-copy mechanism and presented a new zero copy system named ZCopy, which was based on the observation that the metadata around the network data will usually get mutated. Experiments with two applications on an Intel machine show that ZCopy outperforms vanilla Linux for sending a relative large network data package.

In our future work, we plan to extend our work in two directions. First, though we focus specially on web-caching applications in this paper, ZCopy places little constraints on applications and is applicable to other networking applications. We plan to study and evaluate the performance benefit of ZCopy on other network-intensive applications. Second, ZCopy was evaluated using a single core. We plan to extend the ZCopy to efficiently run on multicore machines.

References

[1] Average web response size. http://www.httparchive.org/.

[2] Infiniband. http://www.infinibandta.org/.

[3] LibMemcached. http://libmemcached.org/.

[4] Varnish web cache system. https://www.varnish-cache.org/.

[5] J.C. Brustoloni and P. Steenkiste. Effects of buffering semantics on i/o performance. In *Proc. OSDI*, 1996.

[6] Jerry Chu. Zero-copy tcp in solaris. In *Proc. Usenix ATC*, 1996.

[7] P. Druschel and L.L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proc. SOSP*, pages 189–202. ACM, 1993.

[8] R. LERNER. Memcached integration in rails. *Linux Journal*, 2009.

[9] L. McVoy. The splice i/o model, 1998.

[10] myricom. Myrinet. http://www.myricom.com/scs/myrinet/overview/.

[11] V.S. Pai, P. Druschel, and W. Zwaenepoel. Io-lite: a unified i/o buffering and caching system. *ACM TOCS*, 18(1):37–66, 2000.

[12] J. Pasquale, E. Anderson, and P.K. Muller. Container shipping: operating system support for i/o-intensive applications. *Computer*, 27(3):84–93, 1994.

[13] S. Schneider, C.D. Antonopoulos, and D.S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proc. ISMM*, pages 84–94, 2006.

[14] D. Stancevic. Zero copy i: user-mode perspective. *Linux Journal*, 2003(105):3, 2003.

Seagull: Intelligent Cloud Bursting for Enterprise Applications

Tian Guo
UMASS Amherst

Upendra Sharma
UMASS Amherst

Timothy Wood
The George Washington University

Sambit Sahu
IBM Watson

Prashant Shenoy
UMASS Amherst

Abstract

Enterprises with existing IT infrastructure are beginning to employ a hybrid cloud model where the enterprise uses its own private resources for the majority of its computing, but then "bursts" into the cloud when local resources are insufficient. However, current approaches to *cloud bursting* cannot be effectively automated because they heavily rely on system administrator knowledge to make decisions. In this paper we describe Seagull, a system designed to facilitate cloud bursting by determining which applications can be transitioned into the cloud most economically, and automating the movement process at the proper time. We further optimize the deployment of applications into the cloud using an intelligent precopying mechanism that proactively replicates virtualized applications, lowering the bursting time from hours to minutes. Our evaluation illustrates how our prototype can reduce cloud costs by more than 45% when bursting to the cloud, and the incremental cost added by precopying applications is offset by a burst time reduction of nearly 95%.

1 Introduction

Many enterprise applications see dynamic workloads at multiple time scales. Since predicting peak workloads is frequently error-prone and often results in underutilized systems, cloud computing platforms have become popular due to their ability to rapidly provision server and storage capacity to handle workload fluctuations. At the same time, many medium and large enterprises have significant current investments in IT data centers that house compute and storage systems. This IT infrastructure is often sufficient for the majority of their computing needs, while offering greater control and lower operating costs than the cloud. However, workload spikes, both planned and unexpected, can sometimes drive the resource needs of enterprise applications above the level of resources available locally. Rather than incurring capital expenditures for additional server capacity to solely handle such infrequent workload peaks, a hybrid model has emerged where an enterprise leverages its local IT infrastructure for the majority of its computing needs, and supplements with cloud resources whenever local resources are stressed.

Employing cloud bursting can save enterprises a significant amount of money. Figure 1 illustrates a scenario where a business typically requires five "extra large" servers for its daily needs, but two days a week experiences a spike up to ten servers. Using Amazon's EC2 Cost Calculator [2], we can see that a hybrid approach is most efficient and lowers costs by up to 29% a year.

This hybrid technique, which is referred to as "cloud bursting", allows the enterprise to expand its capacity as needed while making efficient use of its existing resources. While commercial and open-source virtualization tools are beginning to support basic cloud bursting functionalities [11, 10, 14], the primary focus has been on the underlying mechanisms to enable the transition of virtual machines between locations. These systems leave significant policy decisions in the hands of system administrators to determine when to invoke cloud bursting and which applications to "burst". This may lead to poor choices in terms of minimizing cloud costs or reducing downtime during the transition, especially when there are a large number of diverse applications in the data center and different cloud platform pricing models.

We have developed Seagull to alleviate the above challenges; Seagull dynamically decides which applications can be moved to the cloud at lowest cost, and then performs the migrations needed to dynamically expand capacity as efficiently as possible. By automating these processes, Seagull is able to respond quickly and efficiently to workload spikes.

The first *insight* of our work is that rather than naïvely moving an overloaded application to the cloud, it may be cheaper and faster to move *different* applications and

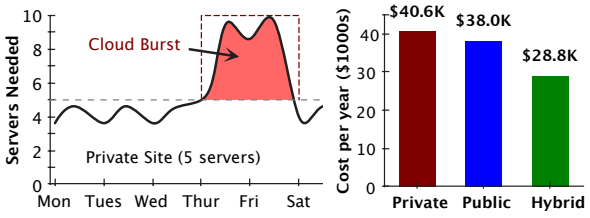


Figure 1: Hybrid clouds can utilize cheaper private resources the majority of the time and burst to the cloud only during periods of peak demand, providing lower cost than exclusively private or public cloud based solutions.

then assign the freed-up server resources to the overloaded application. Bursting an application to the cloud involves copying its disk image and any application data. Since this disk state may be large, a pure on demand migration to the cloud may require hours to copy this large amount of data. The second *insight* of our work is that periodic background precopying of disk snapshots of candidate applications can significantly reduce the cloud bursting latency–since only the incremental delta of the disk state needs to be transferred to reconstruct the disk image in the cloud.

Our paper makes several contributions: **(i)** a placement algorithm that determines which applications should be moved to minimize cost; **(ii)** a precopying algorithm that decides which applications should be proactively replicated to the cloud to enable much faster VM migrations; and **(iii)** a prototype of Seagull and an experimental evaluation of it on a Xen-based local data center and the Amazon EC2 cloud platform. We show Seagull's placement algorithm can make intelligent decisions about which applications to move, lowering the cost of resolving an overloaded large scale data center by over 45%, while precopying significantly lowers burst time with only a modest increase in cost.

2 System Model and Problem Statement

Seagull seeks to enable more agile cloud bursting that can respond to moderate workload spikes within hours or even minutes. We assume that each application is composed of one or more virtual machines that are housed in private data centers, which offer mechanisms for dynamic scaling of server capacity. We assume that applications support either or both of the following mechanisms to scale capacity: i) horizontal scaling: additional replicas are started on demand to increase the capacity ii) vertical scaling: an application's VM is allocated more resources such as more CPU cores. For simplicity, we assume access to a workload forecaster that can predict when a data center is becoming overloaded. We also assume that public cloud follows a resource pricing model

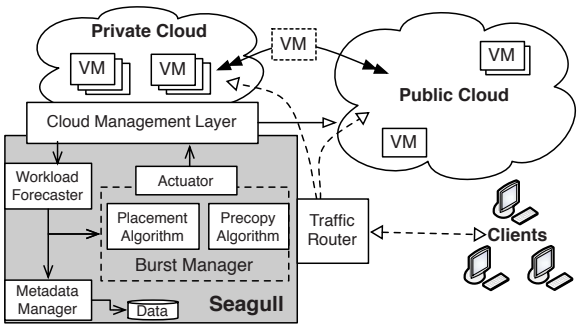


Figure 2: Seagull architecture

similar to Amazon EC2.

We have designed Seagull to both automate and optimize cloud bursting tasks within this setting. In this work, we focus on answering the following two questions: i) Which applications to cloud burst so that cloud server and I/O costs are optimized? ii) How to use judicious precopying to achieve the tradeoff between cloud bursting latency and cost?

3 Placement & Precopying in Seagull

Seagull is composed of multiple interacting components as shown in Figure 2. The *Cloud Management Layer* offers a common interface to interact with both the private and public clouds. The actions performed by this layer are determined by the *Burst Manager*, which is responsible for important decisions about application placement, bursting, and precopying. This section describes the placement and precopying algorithms used by Seagull.

Intelligent Placement The intuition behind the placement algorithm is to maximize the utilization of local resources, which are cheaper than public resources, and migrate the cheapest applications when local resources are insufficient to handle the overload. To do so in a cost-effective manner, the algorithm greedily picks those applications to move that free up the most units of local resources relative to their cost of running in the cloud.[1]

To determine which applications should be moved, we assume the duration of the workload spike, L, and the desired capacity C for each virtual machine are known. Note that C is a vector representing the CPU, disk, network and memory capacity needs of each VM. We define the cost of bursting an application, say A, that is composed of n virtual machines in terms of the cost of transferring its memory and storage, storing the data, and then

[1] Additional administrative criteria such as security policies may also preclude some applications from being valid cloud burst targets; we assume that system administrators provide this information as a cloud bursting black list.

running it in the public cloud:

$$Cost = \sum_{j=1}^{n} C\_tran_j + C\_stor_j + C\_run_j * L, \quad (1)$$

where C_tran_j and C_stor_j are calculated based on the amount of data that must be transferred and stored in the cloud to run the j^{th} VM of A (i.e. VM_j); C_run_j is determined based on the capacity requirements, C, of the virtual machine (e.g., the number of cores it requires) and must be multiplied by L to account for the length of time the VM would need to remain in the cloud before the workload spike passes. We sum the cost across all VMs in the application to account for the constraint that all virtual machines that comprise an application be grouped together either in the local data center or on the cloud. Notice that we can easily plug in different cost functions to account for different pricing models and scenarios such as deploying VMs from the same applications across different data centers.

We can use Equation 1 to calculate the cost of bursting the overloaded application, however, Seagull must decide whether to simply move the overloaded application itself, or to find a different set of applications that can be moved more cheaply in its place. To this end, Seagull must consider each of the VMs that make up the overloaded application and see if there is a way to meet their resource requirements in the local data center by either local reconsolidation or selecting one or more different applications to burst.

The virtual machines of the overloaded application are considered in decreasing order of their resource requirements. For each of these virtual machines, Seagull considers the potential hosts in the local data center sorted by two criteria: 1) their free capacity in descending order and 2) the total cost, in increasing order, of moving all *applications* (including related VMs) on the host to the cloud. The first criteria biases Seagull towards utilizing the free capacity in the local data center first, potentially reducing the number of applications that need to be moved to the cloud. The second criteria ensures that hosts running low cost applications are considered first.

When hosts have been sorted in this way, the algorithm considers the first host and attempts to decide if a set of VMs on that host can be moved in order to create space for the overloaded VM. Each virtual machine, VM_j on the host is ranked based on: $num\_cores_j/Cost$, where *Cost* is the cost of moving the full application that VM_j is part of, and num_cores_j is the number of CPU cores currently in use by the virtual machine. The VMs on the host are considered in decreasing order of this criteria, and the first k VMs are selected such that the free capacity they will generate is sufficient to host the overloaded virtual machine. The intuition behind this greedy heuristic is that it *optimizes the amount of local capacity freed per dollar spent running applications in the cloud.*

Each of the overloaded applications is considered for bursting using this metric. When a solution is found, the total cost of moving all of the marked applications is compared to moving just the overloaded application; the cheaper of the two options is chosen in each case.

Opportunistic Precopying In general, an application's state may be very large. Migrating all of this data at cloud bursting time can take hours or even days, significantly reducing the agility with which a data center can respond to rising workloads.

Seagull performs precopying by transferring an incremental snapshot of a virtual machine's disk-state to the cloud. Seagull's precopying technique must make two important decisions: i) *which* applications to precopy, and ii) *how frequently* to precopy each one. Each of these decisions leads to a cost-benefit tradeoff. The larger the set of candidate applications chosen for precopying, the greater the chances Seagull's cloud bursting algorithm will pick one of the precopied applications to burst to the cloud when the peak workload arrives, increasing the agility of the system to respond to local stress. Similarly, the more frequently each application is precopied to the cloud, the smaller the delta will be, leading to a smaller bursting latency. Thus a careful choice of the candidate set of applications to precopy and precopying frequency can both reduce the overheads. We have implemented a strategy that computes a set of candidate applications to balance the benefits of precopying against its cost, and also two baseline strategies for comparison.

Our cost-benefit tradeoff strategy first generates an *overload list*, i.e. a list of applications likely to become overloaded[2]. Seagull then runs its cloud bursting algorithm, from the previous section, in an *offline* mode over the *overload list*. That is, for each application A on the overload list, Seagull runs its algorithm to see which application(s) get chosen for bursting if A were to become overloaded. These applications form the *precopy list*.

The disk state of applications in the *precopy list* is replicated to the cloud based on a frequency strategy. In the simplest case the precopy frequency can be chosen statically—say once a day or once a week. However, Seagull can analyze the write rates to the virtual disks to "tune" the precopy frequency for each application in the list, managing overall cloud costs while retaining agility.

The two other strategies that we use as baselines for our comparitive evaluation are: i) *Random Precopying:* selects a random set of applications to be precopied based on the maximum expected overload (e.g., ran-

[2]Seagull can generate such a list based on the history of prior cloudburst instances and system administrators can alter it, based on their expert knowledge of which overload scenarios are still likely in the future.

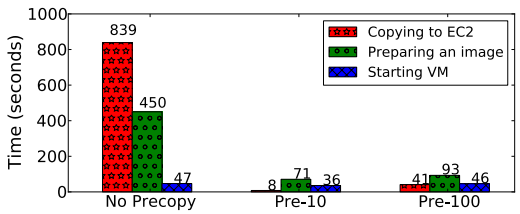


Figure 3: Without precopying, a cloud burst can take tens of minutes to copy data.

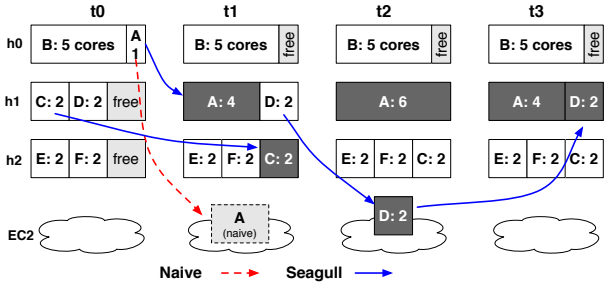


Figure 4: The naïve approach uses only one migration, immediately moving A from h_0 to the cloud. Seagull initially avoids any cloud costs by rebalancing locally, and is able to move back from the cloud sooner than the naïve approach.

domly precopy 20% of the data center's applications). ii) *Naïve Precopying:* selects the set of applications that is predicted to become overloaded for precopying.

4 Experimental Setup and Evaluation

We have implemented Seagull's placement and precopying algorithms as modules that extend the OpenNebula cloud management software. We have created a private cloud environment on a lab cluster using OpenNebula over the Xen-hypervisor and used Amazon EC2 as our public cloud. We use three applications, TPC-W, Wikibooks and CloudStone for our evaluation. We have created private-cloud as well as public-cloud appliances for each of these three applications and their respective client applications. An appliance instance will create the virtual machine(s) which house the complete application. We warm up each application, using its clients, for two minutes before collecting data.

Cloud Bursting Time The total time to perform a cloud burst can be decomposed into three major parts: copying data to the cloud, preparing an application image, and booting up the virtual machine. To measure each of these components, we migrate a virtual machine running the CloudStone application with a disk-state size of 5GB.

As shown in Figure 3, the total time to migrate an application with even a very small 5GB disk state, is 1336 secs ($\sim$ 22 mins); this clearly illustrates the need for precopying in real applications that may have ten or more times as much state. We next precopy the application and reduce the delta (i.e. difference between the original and precopied snapshot) to 10MB or 100MB; the total time to burst the application significantly reduces to less than 200 secs for a delta of 100 MB. Note that as delta reduces the image preparation time and boot time start to flatten around 120 secs and become the prime component of total bursting time.

Placement Algorithm In this experiment, we analyze the placement efficiency of Seagull compared to a naïve algorithm (which always cloudbursts the overloaded application), in a small scenario that demonstrates the intuition behind Seagull's decision making. We show that when a hotspot occurs, Seagull is able to make better use of local resources as well as pick cheaper applications to move to the cloud.

We use three 6-core physical servers, each hosting a pair of different applications: TPC-W (VMs A and D) Wikibooks (B, E), and CloudStone (C, F). Each application is running inside a single VM and can be scaled up vertically. The initial arrangement of applications and the number of cores dedicated to each is shown under t_0 in Figure 4. To simplify the scenario, we assume that all applications have identical storage requirements.

We change application A's workload every hour (marked by instants t_i, where $i = 1 \ldots 3$) such that its CPU requirement increases to four cores, then six cores, before falling back to four cores at t_3. To eliminate the impact of prediction errors in this experiment we assume a perfect forecaster.

Results: When Seagull detects the first upcoming workload spike at t_1, it attempts to resolve the hotspot by repacking the local machines, shifting application C to h_2 and then moving A to h_1 at effectively no cost. In the naïve solution, application A is cloud burst to EC2 directly without considering local reshuffling.

In the workload's second phase, Seagull migrates a cheaper application, D, to EC2 since the local data center could not provide enough capacity needed for A. On the other hand, the naïve algorithm had already moved A to the cloud, so it simply allocates extra resources to it making it more expensive.

Eventually, the workload spike for application A passes, Seagull migrates D back to the local data center while the naïve algorithm, lacking the ability to perform local reshuffling, still needs to keep A in the cloud, wasting more money.

Time and Monetary Cost: The use of local resources in Seagull allows it to respond to overload faster than the naïve approach. Figure 5 shows the amount of time spent by each approach to resolve the hotspots at each measurement interval; note that for both systems we pre-

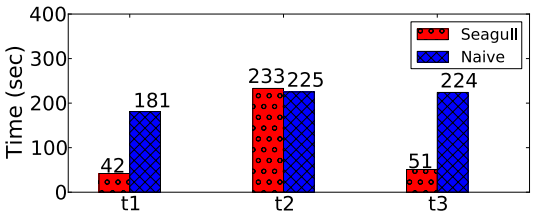


Figure 5: Seagull uses local, live migrations at t_1, and benefits from reverse pre-copying at t_3, substantially reducing the time spent at each stage compared to naïvely cloud bursting at t_1 and restarting instances at t_2 and t_3.

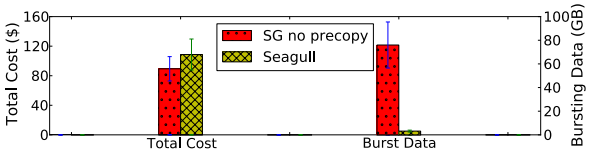


Figure 6: Precopying causes a marginal increase in cost, but a dramatic reduction in burst time.

copy all applications once to the cloud before the experiment begins. Seagull is substantially faster because it uses only a local, live migration at t_1 whereas the naïve approach approach requires a full cloud burst. Subsequent actions performed by Naïve also incur substantial downtime since VMs must be rebooted in the cloud to adjust their instance type to obtain more cores. Seagull's migration back from the cloud at t_3 is also quite fast because it does not require the full image registration process needed for moving into the cloud. Most importantly, the fact that Seagull only requires a virtual machine in the cloud for the hour starting at t_2 means that it pays 30% less in cloud data transfer and instance running costs.

Precopying Algorithm In order to study the impact of precopying, we conducted an experiment by simulating a data center comprised of 200 quad-core hosts and populated it with three types of applications with different disk size and update rate[3] .

We first instruct Seagull to precopy 60 applications to the cloud, and then simulate a hotspot scenario where 30% of the data center becomes overloaded. With this level of precopying, Seagull was able to resolve the hotspot solely by bursting applications which had already been precopied, dramatically reducing the cloud burst time compared to an approach with no precopying. Figure 6 show a modest 22% increase in cost due to precopying, but a substantial 95% saving in data transfer. However, this trade-off can be further tuned based on the precopying algorithm and parameters.

Comparisons: We next evaluate the effectiveness of Seagull's intelligent precopying strategy compared to the random (*SG-random*) and naïve precopying strategies at a larger scale. We use Seagull's placement algorithm to determine the total burst cost when using each of these precopying techniques. We study the decisions made when the level of overload in the data center increases from 10 to 30 percent. Figure 7 presents the average performance of these three strategies when the simulation is repeated 40 times for each level of overload.

In Figure 7(a), Seagull achieves the lowest precopying cost across all overload levels. The benefits of Seagull increase with rising overload levels, and it is able to lower precopying costs by up to 75%. The naïve approach shows the highest cost because there are often applications which can be precopied more cheaply than those which are expected to become overloaded.

Figure 7(b) shows the total cost including both precopying and cloud bursting. Seagull reduces the cost by 45% compared to the Naïve approach because naïvely running the overloaded applications in the public cloud is more costly. SG-Random and Seagull have similar total cost because they select the same applications to burst.

Figure 7(c) shows the total amount of data sent during cloud bursting, which can be used as a proxy for total burst time. Our intelligent precopying strategy far outperforms SG-Random because the latter has a poor chance of precopying the applications that will be selected by the placement algorithm. The naïve algorithm precopies and bursts the same applications, giving performance similar to Seagull, although at higher cost.

5 Related Work

Cloud Computing covers a wide range of types of systems; in this work we focus on Infrastructure as a Service (IaaS) platforms such as Amazon's Elastic Compute Cloud. Armbrust et al. provide a survey of cloud computing [1], and specifically list "scaling quickly" as one of the key opportunities in cloud computing. Many recent projects automate virtual machine or storage migration to balance the CPU, memory, or I/O loads within a single data center [6, 13]. It is increasingly common for businesses and service providers to own multiple data centers, so managing resources across data centers is a growing challenge [12, 4]. We believe that operational expense will naturally expand the automated resource management techniques to include cross data center management approaches like cloud bursting as data centers become connected by increasingly high bandwidth links.

Cloud Bursting was first proposed by Amazon's Jeff Barr as a way to allow enterprises who already own significant amounts of IT infrastructure to still make use of the cloud during periods of high demand [5]. Re-

[3]To eliminate the impact of Seagull's local reshuffling on precopying efficiency, we assume that the data center runs only horizontal-scaling applications, preventing the need for local reconsolidating.

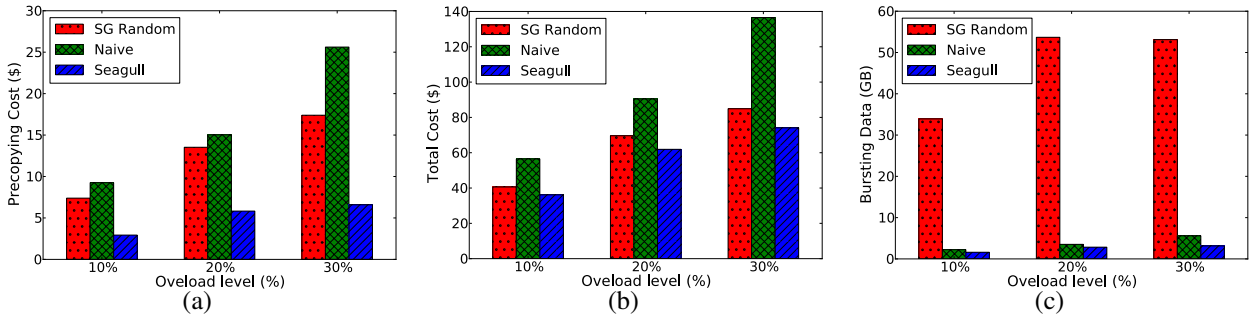


Figure 7: Intelligent precopying reduces total cost and data transferred by over 45% compared to the naïve algorithm.

searchers have been investigating the potential economic savings by using cloud bursting in specific domains such as medical image processing [8] and publishing [7].

Live VM migration over WAN attempt to seamlessly move the memory and storage of a virtual machine between data center sites, usually by building upon the existing LAN migration tools included in modern hypervisors [3]. Alternatively, storage migration tools move only the disk state of applications [9]. Our prototype focuses on storage migration due to limitations of current cloud platforms; however, we note that Seagull could easily be enhanced to support full VM live migration.

6 Conclusions and Future Work

Cloud bursting is a technique to dynamically move applications running in a private data center to the public cloud to take advantage of additional resources there. In this work we propose Seagull, a cloud bursting system that efficiently precopies and migrates applications to the cloud when local infrastructure becomes overloaded. This allows Seagull to perform agile provisioning of resources across a private data center and the cloud, resulting in more efficient utilization of local resources while incurring only minimal expense in the cloud.

In future work, we plan to extend our Seagull prototype to provide complete cloud bursting automation. There are a number of challenges remaining, such as determining when to initiate a cloud burst even if accurate prediction models of future workloads are not available, defining standardized cloud interfaces to enable live WAN migration to public clouds, and integrating Seagull's placement algorithm with business policy requirements.

Acknowledgements: We thank our shepherd Pradeep Padala and reviewers for their comments. This research was supported in part by NSF grants CNS-1117221, CNS-0916972, CNS-0855128 and OCI-1032765 and an Amazon AWS grant. Upendra Sharma was supported by an IBM Graduate fellowship.

References

[1] ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A. D., KATZ, R. H., KONWINSKI, A., LEE, G., PATTERSON, D., RABKIN, A., STOICA, I., AND ZAHARIA, M. Above the clouds: A berkeley view of cloud computing. Tech. Rep. UCB/EECS-2009-28, EECS Department, UCB, Feb. 2009.

[2] AWS Economics Center. http://aws.amazon.com/economics/.

[3] BRADFORD, R., KOTSOVINOS, E., FELDMANN, A., AND SCHIÖBERG, H. Live wide-area migration of virtual machines including local persistent state. In *VEE* (San Diego, California, USA, 2007), ACM, pp. 169–179.

[4] BUYYA, R., RANJAN, R., AND CALHEIROS, R. N. Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services. In *International Conference on Algorithms and Architectures for Parallel Processing* (2010).

[5] Cloudbursting - hybrid application hosting. http://aws.typepad.com/aws/2008/08/cloudbursting-.html, Aug. 2008.

[6] GULATI, A., SHANMUGANATHAN, G., AHMAD, I., WALDSPURGER, C., AND UYSAL, M. Pesto: online storage performance management in virtualized datacenters. In *SOCC* (New York, NY, USA, 2011), SOCC '11, ACM, pp. 19:1–19:14.

[7] KAILASAM, S., GNANASAMBANDAM, N., DHARANIPRAGADA, J., AND SHARMA, N. Optimizing service level agreements for autonomic cloud bursting schedulers. In *ICPP Workshops* (2010), pp. 285–294.

[8] KIM, H., PARASHAR, M., FORAN, D. J., AND YANG, L. Investigating the use of autonomic cloudbursts for high-throughput medical image registration. In *GRID* (2009), IEEE, pp. 34–41.

[9] MASHTIZADEH, A., CELEBI, E., GARFINKEL, T., AND CAI, M. The design and evolution of live storage migration in vmware esx. In *USENIX ATC* (Berkeley, CA, USA, 2011), pp. 14–14.

[10] Open Nebula: The Open Source Toolkit for Data Center Virtualization. http://www.opennebula.org.

[11] openstack: Cloud Software. http://www.openstack.org.

[12] ROCHWERGER, B., BREITGAND, D., EPSTEIN, A., HADAS, D., LOY, I., NAGIN, K., TORDSSON, J., RAGUSA, C., VILLARI, M., CLAYMAN, S., LEVY, E., MARASCHINI, A., MASSONET, P., MUNOZ, H., AND TOFFETTI, G. Reservoir - when one cloud is not enough. *Computer 44* (2011), 44–51.

[13] SHEN, Z., SUBBIAH, S., GU, X., AND WILKES, J. Cloudscale: elastic resource scaling for multi-tenant cloud systems. SOCC '11, ACM, pp. 5:1–5:14.

[14] VMware: Public & Hybrid Cloud Computing. http://www.vmware.com/solutions/cloud-computing/public-cloud/products.html .

The Forgotten 'Uncore': On the Energy-Efficiency of Heterogeneous Cores

Vishal Gupta* Paul Brett† David Koufaty† Dheeraj Reddy† Scott Hahn†
Karsten Schwan* Ganapati Srinivasa‡

*Georgia Tech †Intel Labs ‡Intel Corporation

Abstract

Heterogeneous multicore processors (HMPs), consisting of cores with different performance/power characteristics, have been proposed to deliver higher energy efficiency than symmetric multicores. This paper investigates the opportunities and limitations in using HMPs to gain energy-efficiency. Unlike previous work focused on server systems, we focus on the client workloads typically seen in modern end-user devices. Further, beyond considering core power usage, we also consider the 'uncore' subsystem shared by all cores, which in modern platforms, is an increasingly important contributor to total SoC power. Experimental evaluations use client applications and usage scenarios seen on mobile devices and a unique testbed comprised of heterogeneous cores, with results that highlight the need for uncore-awareness and uncore scalability to maximize intended efficiency gains from heterogeneous cores.

1 Introduction

Energy-efficiency remains a critical concern for both mobile devices and server systems. To improve energy-efficiency while providing high-performance, chip vendors have adopted heterogeneous multicore processors (HMPs). Examples include Variable SMP from NVIDIA [1] and Big.LITTLE processing from ARM [4]. This work focuses on HMPs consisting of a mix of cores that expose the same instruction-set-architecture (ISA), but differ in their power/performance characteristics. HMPs make it possible for different applications within a diverse mix of workloads to be run on the 'most appropriate' cores [3, 5, 6, 7]. For example, applications that do not produce a result that is time critical to the user or that are I/O heavy, can be run on low-power small cores, while compute-intensive threads or applications with their output visible to the user, such as browsing, can be allocated to high-performance big cores.

Previous work on heterogeneous processors has primarily focused on core power [5, 7], but modern multicore processors also contain *uncore* subsystem (see Figure 1), with components like the last level cache, integrated memory controllers, etc. With growing cache sizes, increasing complexity of the interconnection network, various core power optimizations, and the integration of SoC (system-on-a-chip) components on CPU die, the uncore is becoming a significant power component in total SoC power [8]. For energy-efficient operation, therefore, it becomes increasingly important to account for uncore while executing on heterogeneous cores.

This paper investigates the importance of uncore power on the energy-efficiency of heterogeneous multicore platforms. Unlike previous work on heterogeneous processors focused on server workloads [3, 6, 7], it targets client devices where energy is a premium resource and workload profiles are diverse. Since server workloads are not representative of the usage model of client devices, it characterizes the behavior of a diverse set of real-world client applications which are typical of end-user mobile devices and describes different ways in which they can exploit heterogeneity. Using these workloads, it further analyzes the impact of heterogeneity on workload performance and energy-efficiency, including both core and uncore components.

Experimental evaluations use a unique, experimental, heterogeneous multicore platform, comprised of both high and low power cores operating in a shared coherence domain. Results demonstrate that heterogeneous core architectures can provide significant performance improvements while also lowering energy consumption for a diverse set of applications when compared to homogeneous processor configurations. They also demonstrate that potential savings are strongly affected by the 'uncore' contribution, which motivates the need for uncore-awareness in managing heterogeneous multicore platforms and a scalable uncore design to completely realize the intended gains.

2 Beyond Core: Uncore

2.1 What is uncore?

The uncore is a collection of components of a processor not in the core but essential for core performance. The CPU core contains components involved in executing instructions, including execution units, L1 and L2 cache, branch prediction logic, etc. Uncore functions include the last level cache (LLC), integrated memory controllers (IMC), on-chip interconnect (OCI), power control logic (PWR), etc. as shown in Figure 1. With growing cache sizes and the integration of various SoC components on CPU die, the uncore is becoming an increasingly important contributor to total SoC power.

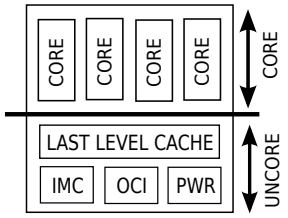


Figure 1: Core and uncore in multicore processors

2.2 Idle State Coordination

Modern multicore processors contain core idle states (C-states) to progressively turn off components in order to conserve power. These C-states are denoted as Cx, where x is a digit. C0 is the active C-state when processor is executing instructions, while a higher numbered C-state (e.g., C2) is a *deeper sleep* state consuming lesser power.

Package		Core 1		
PCx		C0	C1	C2
Core 0	C0	PC0	PC0	PC0
	C1	PC0	PC1	PC1
	C2	PC0	PC1	PC2

Table 1: Core and package idle state coordination

In addition to core C-states, processors also contain package idle states (PCx states) that govern uncore power consumption. These package C-states are related to core C-states in that the processor can only enter a low-power package C-state when all of the cores are ready to enter that same core C-state. Table 1 shows this coordination of core and package idle states for a two-core system with three idle states. The resultant package C-state is always the lower of the two core C-states. Thus, the uncore subsystem remains active and consumes power as long as there is any active core on the CPU.

2.3 Impact of uncore

Figure 2 illustrates the impact of uncore power on the energy consumption of an application executing on heterogeneous cores. A big core running an application finishes its execution faster and enters a low-power idle state. The same application when executed on a small core takes longer (t_{small}) to finish, which also keeps the uncore active for a longer period of time. If uncore power is substantial in comparison to core power, then the energy gains from running on a small core can be strongly affected by the uncore power. For such a system, energy-efficiency gains from small core execution may be offset by the increase in uncore energy consumption due to longer execution time. This observation is in line with prior work that highlights the tradeoff between CPU and system-level power reduction in the context of frequency scaling [9].

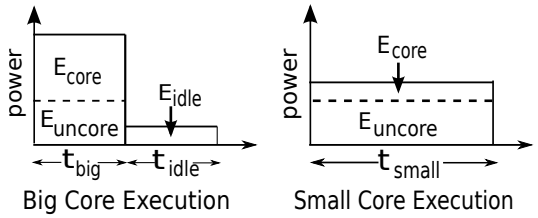


Figure 2: Effect of uncore power on the energy-efficiency of heterogeneous cores

Energy consumption for big and small core execution for such platforms can be modeled using Equations 1 and 2, respectively. Here, E refers to the energy consumed, t denotes execution time, and P_{core} and P_{uncore} represent average core and uncore power, respectively. P_{idle} is the idle platform power, and t_{idle} is the corresponding idle time, as shown in the figure.

$$E_{big} = t_{big} * (P^{big}_{core} + P^{big}_{uncore}) + P_{idle} * t_{idle} \quad (1)$$

$$E_{small} = t_{small} * (P^{small}_{core} + P^{small}_{uncore}) \quad (2)$$

To understand the impact of uncore power, the analysis in Section 4 considers two uncore configurations: fixed and scalable. The fixed uncore configuration uses the same uncore subsystem when executing on either big or small cores. The scalable uncore scenario models an uncore where certain uncore components are turned off or powered down as we move to the small core. For example, fewer memory channels, memory controllers, or a smaller cache can be used with a slow small core that imposes smaller resource requirement on the cache and memory subsystem. Hence, in this case, the uncore power scales along with core power when a workload moves to a different core.

Workload	Description	Metric
browse	loads a set of web-pages at an interval of 3 sec. to emulate user's think time	Load time
javascript	Javascript benchmark performs a series of standard browser operations	Load time
palbum	photo-album application that flips through photographs at 0.5 sec. interval	Load time
mplayer	a H/W accelerated version of mplayer plays an HD movie clip	FPS
mytube	plays an H.264 video inside the browser for 120 seconds	FPS
openarena	plays a benchmarking demo from a 3D first-person-shooter game	FPS
strike	replays a demo session of a web-based 2D game (120 sec.)	FPS
7zip	a parallelized version of 7zip compress a text file using LZMA compression	Time
eclipse	Java based benchmark runs performance tests for the Eclipse IDE	Time
filescan	I/O intensive workload that scans through the Linux source tree	Time
gmagick	GraphicsMagick image editing application is used to resize a set of images	Time
x264	x264 media encoder is used to encode a media file	Time

Table 2: Client workload summary

3 Client Workloads

To assess the viability of using heterogeneity for client systems, we choose a diverse set of real-world applications which are typical of modern end-user devices since prior server-centric research on heterogeneous processors [3, 6, 7] does not directly address the needs and processor usage models seen on client devices. Table 2 provides a summary of the applications used in our analysis and relevant performance metrics. This section categorizes these applications based on their behavior and discusses opportunities for exploiting heterogeneity.

Intermittent Workloads: Client devices like cellphones and tablets are typically powered on for long periods of time, but often perform their heavy processing in short bursts. Web-browsing is an example of such usage, and workloads browse and palbum in Table 2 belong to this category. A timeline trace of IPC (instructions-per-cycle) for the browse workload is shown in Figure 3(a). Idle periods are marked by low IPC periods, while page-loads correspond to spikes in the graph. Since page-loads generate high IPC activity, a big core can be used for rendering the pages and improving page-load performance, while resorting to a small core during low activity periods to conserve power.

Sustained Workloads: Sustained workloads differ from intermittent workloads in that their behavior is uniform over a longer duration. They can be further classified into two sub-categories: sustained-high and sustained-low.

Sustained-low: Client applications like gaming and media playback typically run for a long duration (a few minutes to hours). Moreover, the wide adoption of accelerators allows them to offload significant portions of their computation to accelerators. Figure 3(b) shows the IPC trace of the openarena gaming benchmark. As the observed IPC is low for the application, it can be run on a small core without significant degradation in performance and at lower power (see results in Section 4).

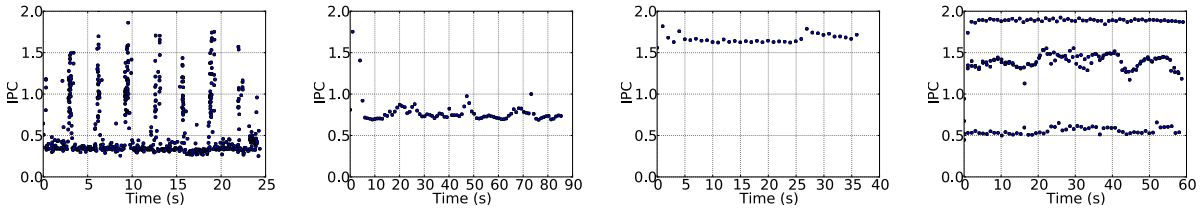
Sustained-high: Mobile devices are also used for compute-intensive tasks such as media encoding, video editing etc. These applications typically have a high IPC (e.g., see x264 encoder in Figure 3(c)), and their performance scales well on a big core. This makes big cores suitable for these applications when they require high performance, e.g., when they are user-facing, while a small core may provide higher energy-efficiency when they run in background mode (e.g, virus-scan).

Multi-threaded Workloads: Increasing core count and parallelization of applications on mobile devices present additional opportunities for exploiting heterogeneity. 7zip, gmagick, and eclipse workloads are examples of parallel applications. Similarly, the mytube workload also uses multiple threads for media decoding and rendering. Figure 3(d) highlights heterogeneity in the mytube workload as various threads within the application differ significantly in their IPC. Since such threads differ in their behavior, heterogeneity can be leveraged by appropriate task scheduling.

4 Experimental Evaluation

4.1 Testbed

Our experimental platform consists of a quad-core Intel i7-2600 client processor. To create heterogeneity, we use proprietary Intel tools to defeature a subset of the cores in order to emulate the performance of low-powered small cores [6]. A block diagram of the platform configuration is shown in Figure 4. An on-die graphics processor is used to accelerate graphics workloads. All of the cores operate at a frequency of 2.6 GHz and share an LLC of size 8 MB. All the workloads are run using Linux kernel 3.0 and automated using scripts.



(a) Intermittent (browse) (b) Sustained-low (openarena) (c) Sustained-high (x264) (d) Multi-threaded (mytube)

Figure 3: Diverse client workload profiles (IPC vs. Time)

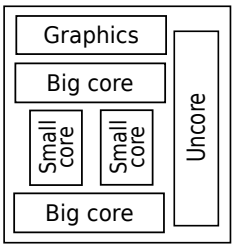


Figure 4: Experimental heterogeneous platform

4.2 Methodology

Experimental evaluation and analysis are carried out as the multiple steps summarized below.

- Each workload is first evaluated on a system configured to use only big cores. Multi-threaded applications are configured for a one to one mapping of threads to big cores.
- Next, the workloads are run using only small cores.
- The metrics collected include: application performance, IPC, LLC accesses, and various core and package C-state residencies.
- With the help of data collected in the previous steps and the power models described in Section 4.3, we calculate the performance improvement and the energy savings of using small vs. big cores.

Our analysis assumes the use of big or small cores for the entire application run. The implementation and evaluation of a dynamic scheduling algorithm for client devices remains part of our future work.

4.3 Power Model

The emulated heterogeneous platform mimics the performance of small cores. However, it does not match the power characteristics of an actual small core built using a different process technology for low power consumption. We, therefore, rely on power models to obtain core and uncore energy consumption.

4.3.1 Core Power

The average power consumption of a CPU core can be modeled using the following equations:

$$P_{core} = R_{active} * P^{core}_{active} + R_{idle} * P^{core}_{idle} \quad (3)$$

$$P^{core}_{active} = C_{dyn} * V^2 * f \quad (4)$$

Here, R_{active} and R_{idle} denote core active and idle state residencies (%), and P^{core}_{active} and P^{core}_{idle} are the corresponding power values. C_{dyn} is the dynamic capacitance, V denotes the operating voltage, and f represents the switching frequency. Big core C_{dyn} is modeled as a function of IPC in Equation 5, as shown and validated by other researchers [10]. Similarly, Equation 6 models the capacitance for a small core having three-times smaller area than that of the big core.

$$C_{big} = 0.499 * ipc_{big} + 0.841 \quad (5)$$

$$C_{small} = 0.472 * ipc_{small} + 0.176 \quad (6)$$

4.3.2 Uncore Power

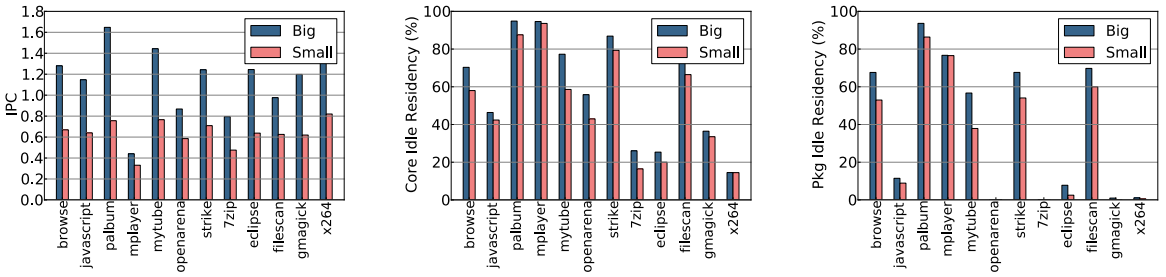
Similar to core power, uncore power is modeled using package idle state residencies (U_x) as shown below:

$$P_{uncore} = U_{active} * P^{uncore}_{active} + U_{idle} * P^{uncore}_{idle} \quad (7)$$

$$P^{uncore}_{active} = P_{wake} + P_{activity} * LLC_{rate} \quad (8)$$

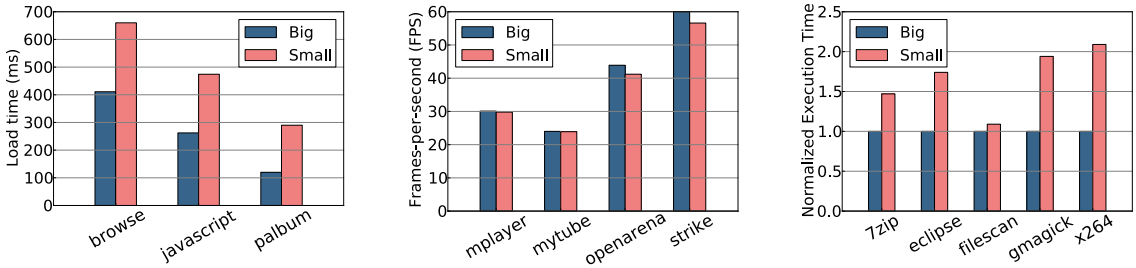
Further, uncore active power (P^{uncore}_{active}) is modeled as a function of LLC activity in Equation 8 where P_{wake} is the fixed power cost of waking up various uncore components, while the $P_{activity}$ component scales with the LLC access rate LLC_{rate} (relative to peak access rate including both cache hits and misses).

The analysis uses a value of 0.9 V for the voltage (V), and frequency (f) is kept at 2.6 GHz. For this platform, the average big core and small core power for all our workloads is obtained to be 2.37 W and 0.95 W respectively. A comparable uncore is modeled using a value of 1.2 W for P_{wake} and $P_{activity}$ in case of a fixed uncore and scaled down to half for a scalable uncore. Core and uncore idle power are assumed to be 0.1 W and a 1.5 W power component is attributed to the on-die graphics processor which also scales with the LLC activity.



(a) Instructions-per-cycle (b) Core Idle Residency (c) Package Idle Residency

Figure 5: A comparison of the behavior of several client workloads on big vs. small cores



(a) Load time (ms) (b) Frames-per-second (FPS) (c) Normalized Execution Time

Figure 6: Application performance comparison on big and small cores

4.4 Results

The results shown in Figure 5 provide a comparison of application behavior on heterogeneous cores. Specifically, they compare average IPC (instructions-per-cycle), core idle residency, and package idle state residency for all of the workloads in Table 2, for big and small core execution. As evident from Figure 5(a), most of the applications observe a significant decrease in their IPC when running on the small vs. big cores. This reduction in IPC results in the small cores being active for longer durations, thereby causing a decrease in core and package idle residency (see Figures 5(b) and 5(c)). Further, many applications are seen to have almost negligible package idle residency. These applications either heavily use the graphics processor (e.g., openarena), or they always keep one of the cores busy (e.g., 7zip, gmagick, x264), and thus do not allow the uncore to enter into an idle state.

The results shown in Figure 6 evaluate the impact on performance of using heterogeneous processors for various client applications in Table 2, categorized by the respective performance metrics. Figure 6(a) compares the average load-time for the browse, javascript, and palbum workloads. We see that the latency is significantly decreased for these applications when using a big core. Thus, a big core provides a notable performance boost for such intermittent applications. In contrast and as depicted in Figure 6(b), when considering the frames-per-second (FPS) metric for various graphics and media applications, we see only minor performance degradation on a small core, at levels not perceivable to end-users. Therefore, they can be run on a small core, to gain potential decreases in energy consumption (discussed further below). The last graph (see Figure 6(c)) compares the normalized execution times for various applications. All of the applications except filescan in this category show a significant improvement in performance with the big core.

Energy savings results computed based on our power models are shown in Figure 7. The figure shows savings for three configurations: core-only savings (C), total SoC-wide savings (C+UC) with a fixed uncore, and with a scalable uncore. As seen in the figure, all of the applications show significant gains on a small core in terms of core energy savings. The palbum application has the lowest savings of 17.58%, while openarena has the largest savings of 52.79%. However, these savings are strongly affected when the power consumption of the uncore is taken into account. Some applications even exhibit negative energy savings. On the other hand, when a scalable uncore is used, these savings increase and become comparable to core-only energy savings. Further, Figure 8 shows the relative contribution of core and uncore energy consumption for all the applications dur-

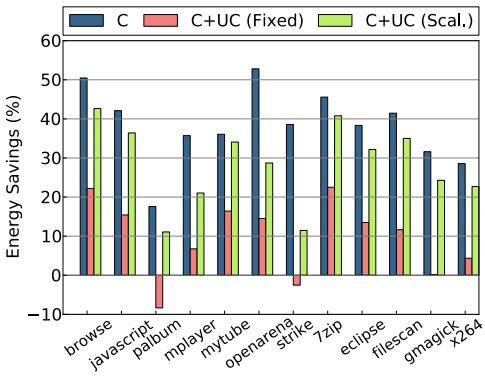


Figure 7: Energy savings of small core execution over big cores for three configurations: core-only savings (C), SoC-wide savings (C+UC) with a fixed uncore, and with a scalable uncore

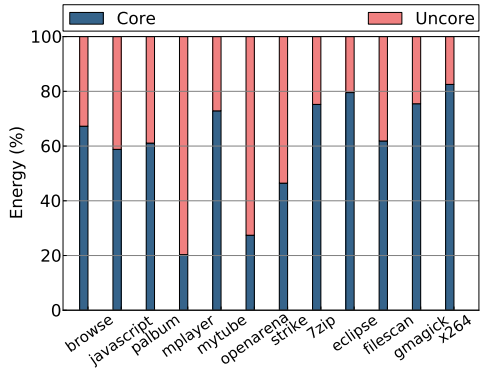


Figure 8: Core and uncore energy contribution

ing big core execution, on a fixed uncore configuration. These results include graphics power in the uncore component. As evident, CPU-intensive applications (e.g., 7zip, gmagick, x264) show a significant core power component, while the uncore fraction dominates for other applications like openarena and mplayer. These results not only demonstrate the importance of taking uncore power into account for scheduling operations, but they also motivate the need for a scalable uncore design to obtain large gains from heterogeneous multicores.

5 Related Work

Substantial prior work has proposed the use of heterogeneous processors to improve the energy efficiency of multicore platforms [3, 5, 7]. Researchers have developed appropriate scheduling algorithms to efficiently run applications on heterogeneous cores [3, 6]. Further, the cost of uncore resources in many-core processors has been modeled and analyzed [8]. In addition, arguments have been made in favor of low-powered cores for the design of datacenters (e.g., FAWN [2]).

In comparison, our work targets client devices where energy is a premium resource, with diverse application behavior and performance metrics. In this context, we highlight the significance of uncore power in total SoC power and analyze its impact on the energy efficiency of several real-world client applications.

6 Conclusions & Future Work

This paper investigates the impact of uncore power on the energy-efficiency of heterogeneous multicore processors for client devices. Using a diverse mix of emerging client applications and an experimental heterogeneous platform, we show that heterogeneous core architectures can provide significant performance and energy gains over homogeneous configurations for client devices. Further, we highlight the growing importance of uncore power with respect to total platform power consumption, thereby motivating the need for uncore-awareness and a scalable uncore design for energy-efficient execution on heterogeneous multicore platforms.

Our future work is investigating client-centric energy-aware scheduling algorithms for heterogeneous multicores. Another interesting venue for research would be to investigate the ideal ratios between the number of big and small cores for different client systems.

References

[1] Variable SMP: A multi-core CPU architecture for low power and high performance. White paper, NVIDIA Corporation, 2011.

[2] ANDERSEN, D. G., FRANKLIN, J., KAMINSKY, M., PHANISHAYEE, A., TAN, L., AND VASUDEVAN, V. FAWN: a fast array of wimpy nodes. In *Proceedings of the SOSP* (2009), ACM.

[3] FEDOROVA, A., SAEZ, J. C., SHELEPOV, D., AND PRIETO, M. Maximizing power efficiency with asymmetric multicore systems. *Commun. ACM 52*, 12 (Dec. 2009), 48–57.

[4] GREENHALGH, P. Big.LITTLE processing with ARM CortexTM-A15 & Cortex-A7. White paper, ARM, Sept 2011.

[5] HILL, M. D., AND MARTY, M. R. Amdahl's law in the multicore era. *Computer 41*, 7 (July 2008), 33–38.

[6] KOUFATY, D., REDDY, D., AND HAHN, S. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th EuroSys* (Paris, France, 2010), ACM, pp. 125–138.

[7] KUMAR, R., FARKAS, K. I., JOUPPI, N. P., RANGANATHAN, P., AND TULLSEN, D. M. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *36th MICRO* (San Diego, CA, USA, 2003), pp. 81—-.

[8] LOH, G. H. The cost of uncore in throughput-oriented many-core processors. In *In Proc. of Workshop on Architectures and Languages for Throughput Applications (ALTA)* (2008).

[9] MIYOSHI, A., LEFURGY, C., VAN HENSBERGEN, E., RAJAMONY, R., AND RAJKUMAR, R. Critical power slope: understanding the runtime effects of frequency scaling. In *ICS* (2002).

[10] SPILIOPOULOS, V., KAXIRAS, S., AND KERAMIDAS, G. Green governors: A framework for continuously adaptive DVFS. In *Green Computing Conference (IGCC)* (July 2011).

Software Techniques for Avoiding Hardware Virtualization Exits

Ole Agesen
VMware
agesen@vmware.com

Jim Mattson
VMware
jmattson@vmware.com

Radu Rugina
VMware
rrugina@vmware.com

Jeffrey Sheldon
VMware
jeffshel@vmware.com

Abstract

On modern processors, hardware-assisted virtualization outperforms binary translation for most workloads. But hardware virtualization has a potential problem: virtualization exits are expensive. While hardware virtualization executes guest instructions at native speed, guest/VMM transitions can sap performance. Hardware designers attacked this problem both by reducing guest/VMM transition costs and by adding architectural extensions such as nested paging support to avoid exits.

This paper proposes complementary software techniques for reducing the exit frequency. In the simplest form, our VMM inspects guest code dynamically to detect back-to-back pairs of instructions that both exit. By handling a pair of instructions when the first one exits, we save 50% of the transition costs. Then, we generalize from pairs to *clusters* of instructions that may include loops and other control flow. We use a binary translator to generate, and cache, custom translations for handling exits. The analysis cost is paid once, when the translation is generated, but amortized over all future executions.

Our techniques have been fully implemented and validated in recent versions of VMware products. We show that clusters consistently reduce the number of exits for all examined workloads. When execution is dominated by exit costs, this translates into measurable runtime improvements. Most importantly, clusters enable substantial gains for nested virtual machines, delivering speedups as high as 1.52x. Intuitively, this result stems from the fact that transitions between the inner guest and VMM are extremely costly, as they are implemented in software by the outer VMM.

1 Introduction

Early x86 processors were not classically virtualizable because some sensitive instructions were not amenable to trap-and-emulate style execution [14]. On such processors, efficient virtualization required the use of *binary translation* to handle all supervisory level code. The binary translator replaced the sensitive guest instructions with the appropriate code to emulate their behavior in the context of the virtual machine (VM). Most contemporary x86 processors now offer *hardware virtualization* (HV) extensions [3, 10]. These processor extensions introduce a new operating mode, *guest mode*, specifically designed to support VM execution using a trap-and-emulate approach. Binary translation is no longer necessary.

While the processor operates in guest mode, most instructions execute at or near native speed. When the processor encounters an instruction or event of interest to the *virtual machine monitor* (VMM), it *exits* from guest mode back to the VMM. The VMM emulates the instruction or other event, at a fraction of native speed, and then returns to guest mode. The transitions from guest mode to the VMM and back again are high latency operations, during which guest execution is completely stalled.

The first generation of processors with HV support exhibited lackluster performance, largely due to high exit latency and high exit rates [1]. However, from the Pentium 4 (Prescott) to the second generation Intel Core Family processor (Sandy Bridge), hardware advances and microcode improvements have reduced exit latencies by about 80%; see Table 1. Moreover, hardware support for MMU virtualization has reduced the exit rate for many workloads by as much as 90%.

Microarchitecture	Launch Date	Cycles
Prescott	3Q2005	3963
Merom	2Q2006	1579
Penryn	1Q2008	1266
Nehalem	3Q2009	1009
Westmere	1Q2010	761
Sandy Bridge	1Q2011	784

Table 1: Hardware round-trip latency.

While hardware implementors have gradually im-

proved exit latencies and rates over the past six years, there is still room for further improvement. The main contributions of this paper are a set of software techniques that, primarily, aim to eliminate exits and, secondarily, allow for somewhat faster handling of exits. The software techniques grew out of our previous work on use of binary translation for virtualization and exploit the observation that exits frequently exhibit temporal locality to reduce exit rate. We identify consecutive *pairs* of instructions that would normally cause back-to-back exits and generate a combined translation to circumvent the second exit. More generally, we identify *clusters* of instructions that would normally cause multiple exits and translate them together to avoid all of the exits save the first. As these translations grow more sophisticated, we expose more opportunities to reduce the exit rate.

The rest of this paper is organized as follows. First, Section 2 describes software techniques for optimizing single instruction exit handling. Our approach draws upon previous experience with using binary translation for efficient instruction simulation. The subsequent sections describe improvements by going beyond single instruction exit handling: Section 3 generalizes to two back-to-back instructions, Section 4 further generalizes to multiple statically recognizable instructions, Section 5 adds use of dynamic information, and Section 6 discusses opportunities involving control flow. Section 7 discusses translation reuse safety. Section 8 applies the techniques to nested virtualization. Finally, Section 9 presents an overall performance evaluation, Section 10 presents related work, and Section 11 offers suggestions for promising future work and our concluding remarks.

2 Exit Handling Speedup

Recent x86 processors from both Intel and AMD provide hardware support to assist a VMM run a virtual machine. Both implementations allow the VMM to directly execute the guest instruction stream on the CPU using guest mode, which disallows operations that require VMM assistance. If the guest attempts such an operation, an exit occurs in which the CPU suspends guest execution, saves the guest context in a format accessible to the VMM, restores the context of the VMM, and starts executing the VMM. The VMM is then responsible for driving guest execution forward past the operation that caused the exit and then resuming direct execution in guest mode. The challenge for the VMM is to handle such exits quickly.

Handling an exit involves driving guest progress forward past the event that caused the exit. Typically this requires interpreting a single guest instruction. Naturally, this involves knowing what the instruction is. In many cases the only way of determining the instruction from the guest context is to take the guest's instruction pointer (`%rip`), read guest memory (while honoring segmentation and page tables), and decode the instruction bytes. This is a complex and time consuming process. In some cases, which vary from CPU to CPU, hints in the guest context suffice to permit the VMM to step the exiting instruction without directly decoding it. However, even on new CPUs a high percentage of the dynamic exits do not provide enough information.

To address such decoding overhead, the VMM can cache decoded instructions keyed by guest `%rip` (possibly combined with other easy to access guest state). When an exit occurs, we can hash `%rip`, find the pre-decoded instruction, verify that the guest still contains the same raw instruction bytes as the cache, and proceed to step the instruction using the cached information.

Years of working on virtualization using a binary translator inspired us to take this decoded instruction caching a step further. Rather than just creating a cache of decoded instructions, we generate executable code in a translation cache for each instruction we want to cache. Our hash function then provides us with an address that we simply jump to. The code at that location verifies that the pre-decoded instruction information matches before running the translated code to handle the exit.

This arrangement allows us to reduce the cost of interpretation by using a form of *specialization*. For instance, most interpreters start out with a large switch statement based on instruction opcode. But since the instruction is fixed for any given translation, our translation can directly embed the correct handler. In a similar manner we are able to further specialize the translation with information about the addressing mode, which guest physical address contains the instruction bytes, or even predictions based on the past behavior of the instruction. This approach provides us with a fast exit handler specialized for each frequently exiting guest instruction.

3 Handling Exit Pairs

When hardware-assisted page table virtualization is unavailable, exits associated with maintaining shadow page table coherency can be among the most frequent [2]. Typically, 32 bit x86 operating systems use Physical Address Extension (PAE) mode to enable addressing of more than 4 GB of physical memory. In PAE mode, page table entries (PTEs) are 64 bits in size. Most operating systems, including Windows and Linux, update these 64 bit PTEs using two back-to-back 32 bit writes to memory such as the sequence, from a Linux VM, shown below. This results in two costly exits:[1]

```
*   MOV   4(%ecx),%esi ; write top half of PTE
*   MOV   (%ecx),%ebx  ; write bottom half
```

[1] In the examples, exiting instructions are marked with an asterisk.

When we generate the translation for the first exiting instruction we decode the next instruction. If the next instruction can be shown to always access adjacent bytes in memory, we create a *pair* translation which merges both instructions to act as a single 64 bit write to memory. The details of recognizing adjacent writes are straightforward: in the above case, we simply note that the two instructions use the same base register, `%ecx`, with respective displacements (0 and 4) that differ by the width of the memory operand (4 bytes). Other cases, including instructions that combine base and index registers, absolute addresses, and `%rip`-relative addresses follow the same approach.

Combining two instructions into a single block avoids taking the second exit, thereby eliminating half of the hardware overhead. But it also reduces software overheads. Each time a guest writes to a PTE, the VMM must make the corresponding adjustment to the shadow page tables. By treating the pair of guest instructions as a single 64 bit write we can transition the shadow page tables directly to the new state in one step. With a 32 bit Linux guest using PAE, the combined reduction in overheads reduces the time it takes to compile a kernel by 12%.

4 Static Cluster Formation

When generating a translation for an exiting instruction, it is straightforward to decode ahead in the guest instruction stream to look for later instructions that will cause subsequent exits. Our translator scans forward a small number of guest instructions, 16 in our implementation, starting from the exiting instruction. It then analyzes the decoded instructions and tries to form a cluster that covers the stretch of guest code from the first exiting instruction to the last one (and no further). If such a cluster is found within the decoded instructions, a translation is emitted for it.

For example, consider this sequence of 16 bit BIOS instructions:

```
* OUT   %eax,%dx
* OUT   $0xed,%al
  MOV   %dx,$0xcfc
  MOV   %al,%cl
  AND   %al,$0x3
  ADD   %dl,%al
  XCHG  %ecx,%eax
  XCHG  %ah,%al
* IN    %al,%dx
  XCHG  %al,%ah
  XOR   %cl,%cl
  JMP   %bx
```

We took an exit on the first `OUT` instruction. Clearly, it is beneficial to translate (and execute) the second `OUT` as well. Moreover, by executing through six ALU instructions (which never exit) we can reach an `IN`. We call such non-exiting instructions *gap fillers* because they fill the gaps between exiting instructions. In this example, the optimal translation covers the first nine instructions but omits the last three for which no benefits are to be had. At runtime, we take an exit on the first `OUT` and resume after the `IN`, avoiding two out of three exits to net a 3x execution speedup.

Is this case a rare oddity? No, most guests contain dozens if not hundreds of similar basic blocks. For example, Linux kernels of a certain vintage worked around a chipset timing bug by piggybacking an extra `OUT` on every `IN` and `OUT`. For a native execution, the cost of the extra `OUT`, while measurable, is affordable. However, in a VM, the work-around is much more expensive because it doubles the number of `IN`/`OUT` related exits. Here's an example:

```
* IN    %al,%dx
* OUT   $0x80,%al     ; bug workaround
  MOV   %al,%cl
  MOV   %dl,$0xc0
* OUT   %al,%dx
* OUT   $0x80,%al     ; bug workaround
* OUT   %al,%dx
* OUT   $0x80,%al     ; bug workaround
```

With clustering, not only do we overcome the exit count increase due to the bug work-around, but we also avoid individual exits on the three required `IN`/`OUT` instructions.

Our final example comes from Windows XP running PassMark where the guest's context switching code reprograms debug registers (for reasons unknown to us this PassMark process runs under debugger control):

```
* MOVDR %dr2,%ebx
* MOVDR %dr3,%ecx
  MOV   %ebx,0x308(%edi)
  MOV   %ecx,0x30c(%edi)
* MOVDR %dr6,%ebx
* MOVDR %dr7,%ecx
```

In a PassMark 2D run, this cluster and other similar ones, compress 46 million would-be exits down to just 12 million actual exits, improving the benchmark score by 50–80% (depending on the CPU used).

The PassMark example has two memory-accessing gap fillers. When we pull memory accesses out of direct execution and into translated code in the VMM's context, we can no longer use x86 segmentation and paging hardware. Instead, address translation must be done in software, slowing down memory accesses. To prevent gap filler overheads from overwhelming exit avoidance savings, we cap the number of memory accesses between exiting instructions at four.

5 Dynamic Cluster Formation

Instructions of types that always exit are *strongly-exiting*. For our VMM, these instruction include `CPUID`, `OUT` and `HLT`. Capturing strongly-exiting instructions in a cluster simply requires decoding forward from the exiting instruction. However, many exits are caused by *weakly-exiting* instructions: loads, stores and read-modify-write instructions that target either memory with traces (such as page tables) or memory-mapped devices. In these cases, inspection of the instruction itself cannot reliably determine whether it will exit or should be treated as a gap filler. Consider this basic block from SUSE Linux Enterprise Server 10, where we have observed an exit on the first instruction:

```
* MOV   -0x201000(%rax),%edx
  MOV   %eax,-0x7fc4215c(8*%rsi)
  AND   %eax,$0xfff
  SUB   %rax,%rcx
* MOV   -0x200ff0(%rax),%r8d
  MOV   %eax,-0x7fc4215c(8*%rsi)
  MOV   %edx,0x60(%rsp)
  AND   %eax,$0xfff
  SUB   %rax,%rcx
* MOV   -0x201000(%rax),%edi
  MOV   %eax,-0x7fc4215c(8*%rsi)
  AND   %eax,$0xfff
  SUB   %rax,%rcx
* MOV   -0x200ff0(%rax),%edx
  MOV   %rax,0x21f1a1(%rip)
```

It may be possible to prove that a downstream instruction *must* exit by starting from the fact that the initial instruction did so. For example, one could use forward data-flow analysis to find relationships between operands of the first instruction and operands of the downstream instructions. Exit pairs, described in Section 3 are a degenerate and successful example of this type of analysis, but the general problem is much harder and is best solved by approximation.

Instead of attempting static analysis of guest code, we have found that a dynamic prediction-based approach is both simpler to implement and more powerful: it can with high accuracy predict instructions that are likely to exit most of the time, and do so without knowing anything about x86 instruction semantics, 16/32/64 bit code, segmentation, etc. In the above example, the instructions marked with an asterisk were predicted to exit so the optimal translation unit *excludes* the last instruction as it will execute faster directly in the context of the guest.

To obtain a dynamic prediction we defer translation until the third time an instruction exits, handling the first two exits with an interpreter and recording untranslated exiting instructions. Then, when we eventually translate, we will have observed two executions of the downstream

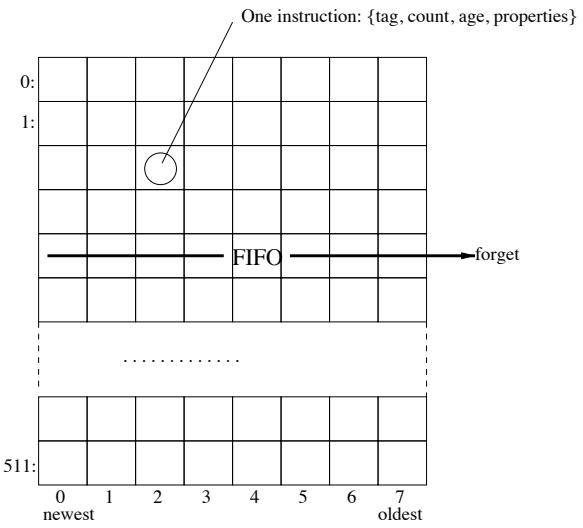


Figure 1: Exit tracking using leakage-rate aware FIFOs.

instructions from the exiting instruction, allowing the use of past behavior as a predictor of the future. This permits us to form clusters that avoid a large number of exits without being too greedy (which would incur overheads from translated execution of non-exiting instructions in a cluster suffix).

The exit-tracking data structure must be compact, as is true for any VMM data structure, and offer efficient guest instruction lookup since it is on the critical path for exit handling prior to translation. A fixed-size hash table can meet these requirements, but it leaves one problem unaddressed: instructions may leak due to finite capacity, and if that happens, how do we ensure that we will eventually get the guest exit working set translated? Failure to "count to three exits" (due to hash table leakage) could leave us perpetually interpreting exits, suffering an unacceptable performance loss. For example, a guest reboot loop where the guest uses address space randomization could cause exit-tracking leakage since each reboot incarnation of the guest would look different and impose a new exit working set upon the VMM. In a pathological case, entries could leak from the hash table so fast that some instructions' execution count fail to reach the translation threshold, forcing the VMM to interpret on every exit and, thus, resulting in severe loss of performance.

Instead of a simple hash table, we combine hashing with an array of 8-deep FIFO queues to track exiting instructions. An exiting instruction is hashed into a 35 bit value (using the program counter, length, and instruction bytes). This 35 bit value is split into a 9 bit index and a 26 bit tag. The index selects a row in a 512-long array of FIFOs where each element in the FIFO has a 26 bit tag, a one-bit saturating counter, a 4 bit relative age (time since last exit), and one or more property bits (see below). Fig-

ure 1 illustrates this data structure. In response to an exit we hash the instruction and traverse the resulting row in the array of FIFOs to look for a matching tag. The first time an instruction exits, we insert it in the FIFO with a count of zero, shifting out the oldest instruction in the FIFO. The second time an instruction exits, we change its count from zero to one. The third time an instruction exits, we translate it.

To guard against the performance cliff that results if we fail to ever translate (interpreting every exit), when we drop the oldest element from the FIFO, we look at its age. If the age is younger than a set threshold (think of it as 10 seconds on a 2 GHz CPU), we must be "leaking entries fast" and in response we force translation regardless of exit count. The highest rate we may interpret without disabling the threshold requirement is $2 * 8$ instructions per FIFO per 10 seconds. Multiplied by the number of FIFOs, we have an affordable maximal rate of interpretation: 820 Hz.

The forced translation may cause loss of optimization opportunities, but it avoids falling over the much deeper cliff of interpreting an unbounded number of exits. In practice, we have not encountered any guest that forces us into immediate translation and the resulting loss of optimization opportunities.

We can go one step further than merely using this FIFO data structure to predict which instructions will exit. Since each instruction will be interpreted at least twice before being translated, we can observe not just the fact that instructions exit but *why* they do so. For example, we could record which device they access (if they are device accessing instructions), or whether they access traced memory. In our experience with a large number of common guest operating systems, any given instruction tends to access either one device or another or page tables, but not a variety of such. This means that for most instructions, we can use our FIFO data structure to issue a reliable prediction that not only will the instruction be likely to exit, but also the reason for it to exit. Even if we don't cluster a particular instruction, knowing the likely cause for it to exit allows us to generate better code. Today, we track accesses to the Advanced Programmable Interrupt Controller (APIC), allowing us to speed up guest interrupt processing by a measurable amount. This same idea could be extended to recognizing accesses to other memory-mapped devices that are deemed sufficiently important, such as network interface cards, SCSI devices, etc.

6 Control-Flow in Clusters

In many cases, exiting instructions are separated by control-flow instructions. Extending clusters from straight sequences of instructions (i.e., basic blocks) to code sequences with arbitrary control-flow can thus further increase the benefits of clustering. However, including control-flow instructions must be done carefully to avoid costs that outweigh the benefits of clustering.

We describe three increasingly powerful approaches. Our implementation gradually evolved through these different methods over time.

6.1 Method 1: No Intra-Cluster Control Flow

The simplest approach allows branch instructions inside clusters but treats all taken branches as cluster termination points. At runtime, a prefix of the cluster will be executed. For instance, the execution of the cluster below would either reach the second exiting `IN` instruction if the `JNZ` branch instruction falls through, or would terminate the cluster in the middle if the branch is taken:

```
* IN    %ax,%dx
  AND   0xd0,$0xffffffeff
  TEST  %ax,$0x20
  JNZ   0x6       --+ M1: terminate if taken
  OR    0xd0,$0x100 |
  MOV   %cx,$0xff <-+ M2: intra-cluster jump
* IN    %ax,%dx
```

6.2 Method 2: Forward Branches

A more powerful approach allows intra-cluster forward branches. At runtime, some of the instructions in the cluster may be skipped if forward branches are taken. In the example above, execution reaches the last exiting `IN` instruction regardless of the path taken at the `JNZ` instruction. Forward control-flow provides a simple mechanism to ensure we bound the amount of time spent outside of direct execution.

To generate efficient code, the translator does not update the virtual `%rip` at each instruction inside the cluster. Instead, it updates it at cluster termination and before calling service routines that may cause early termination. As such, the translation of intra-cluster jumps requires additional *glue* code to adjust the delta `%rip` amount from the source instruction to the target.

6.3 Method 3: Loops

The most general approach is to allow arbitrary branches within a cluster, including control-flow backedges. Although branches to prior instructions in the instruction stream do not necessarily imply loops, in practice all backedges we have encountered correspond to cluster loops. Below is an example of a cluster containing two small loops at the beginning, followed by more exiting instructions:

```
* IN    %al,%dx  <---+
  TEST  %al,$0x8     |
  JNZ   0xfb     ----+
* IN    %al,%dx  <---+
  TEST  %al,$0x8     |
  JNZ   0xfb     ----+
  MOV   %dx,%bx
  MOV   %al,0x18(%esp)
* OUT   %al,%dx
  INC   %dx
  MOV   %al,0x1c(%esp)
* OUT   %al,%dx
  MOV   %al,0x20(%esp)
* OUT   %al,%dx
```

Including loops in clusters has the potential of saving large numbers of exits in a single cluster execution. For instance, during the 64 bit Ubuntu installation a cluster containing loops executes about 20,000 loop iterations in a single runtime instance. However, with this benefit also come two potential dangers: interrupt starvation and performance degradation due to long executions in translated code.

To avoid both of these dangers, we require that each loop iteration checks for pending interrupts and executes at least one exiting instruction. Instead of implementing a full-blown control flow analysis to examine all paths in the cluster, we designed a much simpler, yet effective analysis: for each backedge, we require that the straight list of instructions starting from its target up to the first control-flow instruction encounters both an exiting instruction and an instruction whose translation checks for interrupts. Otherwise the backedge is disabled: the branch instruction is still included in the cluster, but its branch taken path is treated as a cluster termination point. This approach works well in practice: about 84% of all analyzed backedges meet both the interrupt checking and exiting instruction requirements.

But a danger still lurks in the presence of loops that contain merely weakly exiting instructions. Such loops risk degrading rather than improving performance. Consider a loop containing a memory access instruction that we have identified as exiting (using the exit-tracking structure from Section 5). If only a small fraction of the loop iterations touch traced memory, then only a small number of the dynamic instances of that instruction would require exits. Running the entire loop in the cluster may be less efficient because non-exiting memory accesses run faster in direct execution. To contain this risk, we classify cluster backedges as either strong or weak, depending on whether they are guaranteed to reach a strongly exiting instruction or not. We then cap the number of weak backedges traversed per dynamic cluster execution to a small number, 10 in our implementation, while allowing unlimited traversal of strong back edges. This gives us a good balance between achieving most of the benefits and avoiding severe performance degradation in the worst case: if none of the weakly exiting instructions would cause exits, we quickly leave the cluster; if all of the weakly exiting instructions would trigger exits, we only miss 10% of the exit reduction opportunities for that loop.

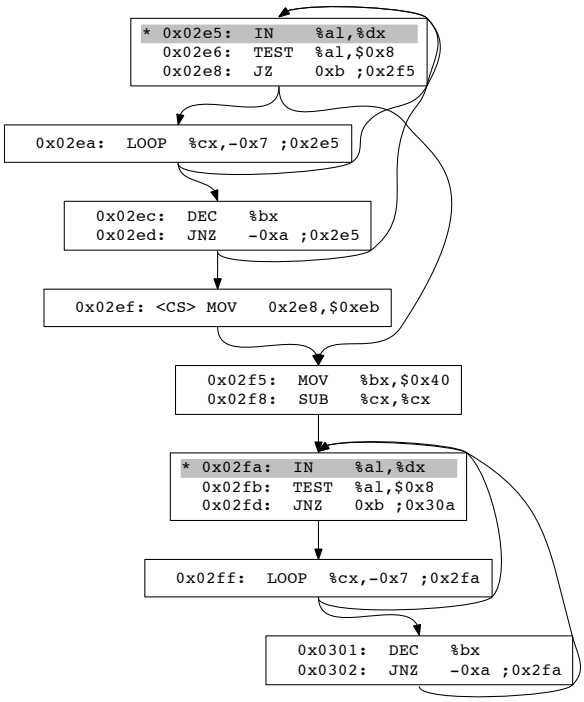


Figure 2: A cluster with complex control-flow from Windows ME. The two exiting instructions are shadowed.

In the absence of loops and backedges, clusters always end with an exiting instruction since going further would yield no benefits. This is no longer true for loops: the translator extends clusters up to the last backedge, if any. In particular, it is possible to form a special case of clusters having a single exiting instruction at the beginning and ending with a backedge to the first instruction, such as this code fragment from Windows 7:

```
* MOV   (%rcx),%rdx <--+
  ADD   %rcx,$0x8      |
  DEC   %r9            |
  JNZ   -0xc       ---+
```

Due to their small size, clusters usually have simple control-flow. But this is not always the case. Control-flow can become fairly complex within less than 16 consecutive instructions, as shown in Figure 2.

7 Checking Code Consistency

Cluster translations are reused every time an exit matches the code address of the cluster. To ensure safety of reuse we must detect cases where the cluster code changes between translation time and use time. This can happen either if a new piece of code gets mapped in at that address, or in the case of self-modifying code.

To detect code changes, each cluster translation begins with a code coherency checking fragment that dynamically checks each byte of the cluster against its corresponding translation-time value. If a mismatch is detected, the existing translation is thrown away and execution falls back to executing just the current exiting instruction. Subsequent exits at the same address will try to form another cluster.

Coherency checks at cluster entry are necessary, but are not sufficient to protect against code modifications. Even if the cluster code matches at the time when an exit is taken, the code may change *during* the execution of the cluster. In other words, the cluster may be self-modifying. We have encountered such clusters in Windows "PatchGuard" code. To address this second issue we have enhanced the translation of memory writes inside the cluster with checks against the pages being accessed. If a cluster tries to write into one of the pages that the cluster belongs to, the cluster execution is terminated. Because clusters contain a small number of adjacent instructions, they span at most two pages. Hence, self-modifying code checks require at most two page checks.

The reader might have noticed that even the cluster in Figure 2 contains self-modifying code. When the `MOV` instruction at address `0x02ef` executes, it changes the opcode of the first branch at address `0x02e8` from `JZ` to `JMP`. Our cluster runtime checks detect this self-modification and immediately terminate the cluster. The next exit at address `0x2e5` will fail its cluster-entry coherency checks, in turn causing cluster re-translation.

8 Nested Virtualization

There is growing interest in *nested virtualization*, where an inner VMM runs inside a virtual machine managed by an outer VMM (see Figure 3). Unfortunately, today's hardware does not provide direct assistance for virtualizing HV, so support for virtual HV must be provided by the outer VMM through software emulation. When a hardware exit occurs, control passes to the outer VMM. The outer VMM must then determine whether it should handle the exit itself, or whether it should forward the exit to the inner VMM [13]. To forward an exit to the inner VMM, the outer VMM must emulate the effects of the exit on the state of the inner VM's virtual CPU. This emulation is extremely slow. With VMware Workstation 8 running on a Sandy Bridge CPU, the virtual hardware round-trip latency for a virtualization exit is 9794 cycles, which is over 10 times worse than the underlying physical hardware; compare with Table 1.

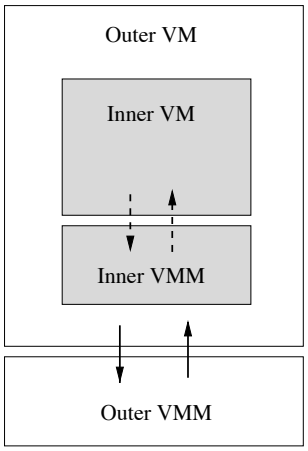


Figure 3: Nested VMs: the inner VM and VMM run inside an outer VM. The solid arrows for the outer VM represent hardware virtualization transitions; the dashed arrows for the inner VM represent emulated hardware virtualization transitions.

Even though the outer VMM gets control of the processor on each hardware exit, we do not try to avoid exits which would have to be forwarded to the inner VMM anyway. When an exit is forwarded to the inner VMM, a large amount of virtual CPU state is modified, including the mappings from linear addresses to physical addresses. Since we do not create translations that span changes in guest paging mode, it is important that the inner VMM exploits all available opportunities for exit avoidance. Every virtual exit that the inner VMM can avoid saves a corresponding hardware exit and the software overhead for forwarding that exit from the outer VMM to the inner VMM.

Exit avoidance in the outer VMM is also significant for nested virtualization. A guest hypervisor often exhibits a higher exit rate than a typical guest, due to frequent modifications of control registers and model specific registers (MSRs), as well as the hardware-assisted virtualization instructions themselves. This is particularly true on Intel hardware. To process an exit, the VMM must access fields in a *virtual machine control structure* (VMCS) that contains the state of the VM. In Intel's implementation, the VMCS is accessed by the new instructions, `VMREAD` and `VMWRITE`, which are strongly-exiting instructions. Typically, a VMM will execute ten or more of these instructions to process a single exit. Thus, a single round-trip from the inner VM to the inner VMM and back again can require ten or more round-trips between the outer VM and the outer VMM, in addition to the emulated exit.

One possible solution to this problem is to paravirtu-

alize VMREAD and VMWRITE [6]. However, clustering in the outer VMM offers some performance benefit without resorting to paravirtualization. For example, in this code sequence from a Hyper-V inner VMM, we save two of three exits:

```
* VMREAD %r9,%rax          ; Exit qualification
  MOV    %eax,$0x2400
* VMREAD %r10,%rax         ; Physical address
  MOV    %eax,$0x4408
* VMREAD %rax,%rax         ; IDT vectoring
```

A clustering-aware inner VMM can help its outer VMM avoid even more exits through dense packing of exiting instructions, *without* creating any incompatibility with unnested execution. This example, from a VMware inner VMM, is executed as part of the exit processing for every (nested) exit:

```
* VMREAD -0x22222a(%rip),%rbx ; RIP
* VMREAD -0x222209(%rip),%rcx ; RSP
* VMREAD %rdx,%rdx            ; RFLAGS
* VMREAD -0x1f229b(%rip),%rbp ; Intr blocking
* VMREAD %rdi,%rdi            ; Exit reason
* VMREAD %rsi,%rsi            ; IDT vectoring
* VMREAD %rbx,%rax            ; CS rights
  ADD    %eax,$0x2
* VMREAD %rbp,%rax            ; SS rights
  TEST   %edi,%edi
  JNZ    0x24
  MOV    %eax,$0x4404
* VMREAD %rax,%rax            ; Interrupt info
```

Each execution of this cluster avoids at least seven hardware exits. By using clustering in the outer VMM and dense packing of exiting instructions in the inner VMM, we have measured a 34% improvement in the time it takes for the inner VMM to process a single strongly-exiting instruction in a nested VM on Intel hardware.

The benefits of clustering in the outer VMM are less significant for nested virtualization on AMD hardware since AMD-V has no equivalent of VMREAD and VMWRITE. However, there are still some opportunities for clustering exits in the outer VMM. Moreover, exit clustering in the inner VMM is just as important on AMD hardware as it is on Intel hardware.

Table 2 illustrates the speedups observed for compiling the Linux kernel in a nested VM with clustering enabled for the inner VMM, the outer VMM, and both VMMs together. Speedups are computed as the ratio of the running time with clusters disabled in both the outer and the inner VM to the running time with clusters. The inner VM runs a 32 bit PAE Linux kernel and the inner VMM uses shadow paging to demonstrate the increased benefits of exit pairs in a nested context. The outer VM runs a 64 bit VMware VMM hosted on a 64 bit Linux OS and the outer VMM uses hardware MMU virtualization.

		Inner			
		Off		On	
		Intel	AMD	Intel	AMD
Outer	Off	—	—	1.20	1.13
	On	1.27	1.06	1.52	1.19

Table 2: Nested kernel compile speedups due to clusters.

9 Evaluation

The techniques described in this paper have been fully implemented in VMware products, including ESX, Workstation, and Fusion. The implementation evolved over several years, starting with simpler methods such as just detecting pairs or consecutive IN/OUT instructions in older releases to full support for clusters with arbitrary control-flow and loops in more recent releases. The implementation has therefore been implicitly validated by years of use in the field.

The implementation comprises approximately 10,000 lines of commented C code. This line-count includes the code to translate handlers for single-instruction exits, but excludes functionality that was already present in our VMM for other reasons, such as the x86 instruction decoder, instruction emitters, the translation cache, and fault handlers.

Earlier sections of this paper show performance results for particular techniques along with the description of the technique. These numbers were harvested at the time we implemented the optimization and focused on that one step only. We now step back and look at the aggregate effects of clustering across a number of workloads, including both regular and nested virtual machines.

Figure 4 shows performance data for several variations of Linux kernel compilation (KC), and a nested VM performing Linux kernel compilation (NKC) as well as PassMark 2D graphics. In each case, we run the workload twice, once without clusters and once with clusters. For the KC workloads, we plot the ratio of running time without clusters to running time with clusters. For the NKC cases, as described in Section 8, the nominator is the running time of the nested setup with clustering disabled both in the inner and outer VMM. For PassMark, where a higher score is better, we plot the inverse ratio of scores, i.e. scores with clusters divided by scores without clusters. A ratio greater than 1.0 therefore indicates a performance improvement.

The PassMark test ran in a Windows XP VM using VMware Workstation. The KC benchmarks ran in a SUSE 10.1 VM (either 32 bit using PAE or 64 bit, as indicated). The compiling VM was configured with 2GB of RAM and measurements were taken after a warm-up run so that the workload could run out of the buffer cache. The NKC benchmarks ran in the same SUSE

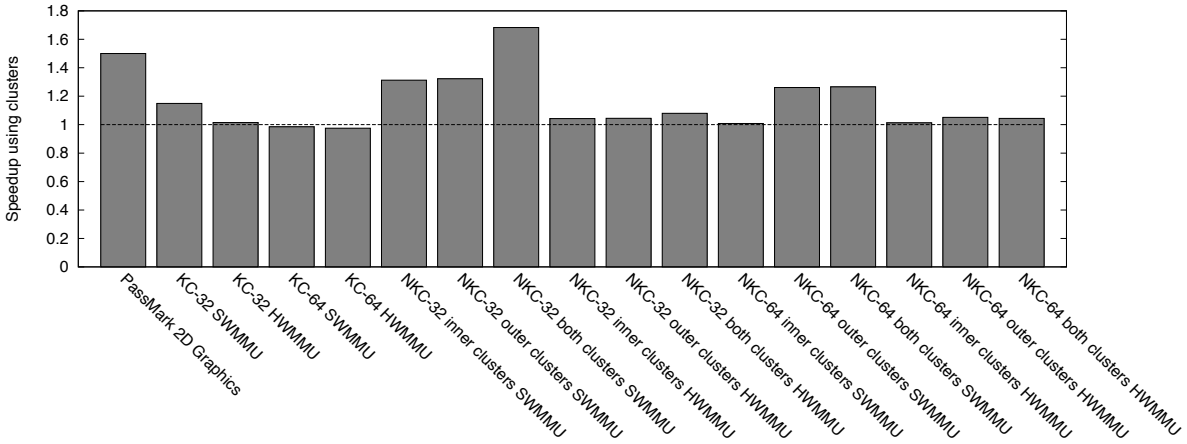


Figure 4: Relative performance improvements using clustering.

10.1 VM (of the indicated code size) running on the same build of VMware Workstation inside an outer 64 bit SUSE 10.1 VM itself running inside VMware Workstation. The outer SUSE 10.1 VM was allocated 4GB of RAM to avoid the need for any disk swapping.

The PassMark result has been previously discussed, and is repeated here for easy comparison, so consider the KC results in the second through fifth column in the figure. We first note that clusters deliver a significant improvement for 32 bit KC when using the software MMU due to dynamically frequent PTE update pairs. The 64 bit KC test, in contrast, shows no gain (the small slowdown is within the noise) as the majority of exits are still due to PTE updates but (1) no PTE pairs exist since a 64 bit VM can (and does) update a PTE with a single 64 bit write and (2) we are unable to cluster from one PTE update to another. While this result may at first blush seem disappointing, the 64 bit KC test runs more than twice as fast as the 32 bit KC test. Put differently, in absolute terms, the 64 bit KC test case has quite good performance from the outset and offers little opportunity for clustering to give it a helping hand. Likewise, when using hardware support for MMU virtualization, clustering provides no benefits whether the guest is 32 bit or 64 bit since this configuration has too low an exit frequency for any exit optimization to matter much.

The rightmost twelve columns in Figure 4 show various nested kernel compile configurations. For NKC workloads we have the option of enabling clustering independently on the inner or the outer VMM. Enabling clustering on just the inner level shows a similar but exaggerated pattern as the non-nested case: 32 bit compiles win significantly while 64 bit compiles show little if any improvement. The larger savings for the nested case result from nested exits being substantially more expensive

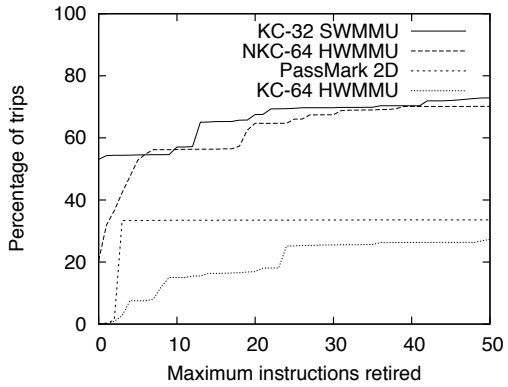


Figure 5: Instructions retired between exits.

than non-nested exits.

Enabling clustering on the outer VMM shows improvements for both 32 bit and 64 bit compiles. Here, instead of avoiding exits from the inner guest, we see benefit from being able to cluster virtualization-related exits made by the inner VMM. This win is visible when using both the software MMU and the hardware MMU, although the win is larger when using the software MMU due to the higher exit rates of the inner guest.

The effects of clustering at the inner and outer levels are not strictly additive but do complement each other as can be best seen by comparing the 32 bit NKC cases.

Clustering fundamentally depends on temporal locality of exits to deliver benefits. With hardware performance counters we can measure the distribution of "distances" between consecutive exits. This data offers an upper bound on the clustering opportunities that exist for an ideal binary translator: one that yields no slowdown over guest mode direct execution and has infinite trans-

lation cache capacity. While an ideal translator does not exist, the distribution still offer insights, including suggesting how close to ideal we can reasonably get with translation units capped at 16 instructions.

Figure 5 shows, for a selection of workloads, the cumulative percentage of trips through guest mode where the number of instructions retired is less than or equal to the indicated maximum. PassMark "jumps" to almost 40% at just two instructions, reflecting the extremely high dynamic frequency of the tight debug-register clusters. Even more impressively, 32 bit KC, with the zero-distance PTE pairs, starts out at over 50% of all exits having zero distance, i.e., they must be pairs.

These graphs show that for many workloads a substantial fraction of exits can be captured with simple clusters of at most 16 instructions. However, this does not mean that all workloads benefit meaningfully from clustering: exit rates must be factored in to obtain the "bottom line." For example, the 64 bit KC, which we previously noted shows no measurable runtime benefit from clustering still has 16% of all exits amenable to clustering, but an insufficient exit frequency for it to matter.

Nested VMs, KC, and PassMark all benefit exceptionally well from clustering, and were motivating cases during development. We looked to VMmark 2.0 for testing against other guests, using only the portions of VMmark that involved actually running VMs and omitting meta-tasks like deploying and moving VMs. Since VMmark runs the workloads for fixed duration (including a ramp-up time), we cannot measure speedup with VMmark. Instead, we looked at the reported throughput scores for the individual VMs. We found that results with and without clustering were about the same, factoring in the noise margin. To determine if this null result is due to clustering being ineffective or due to exits being too infrequent for exit avoidance to matter, we measured exit rates per virtual CPU for the VMmark constituent workloads during their steady state period; see Figure 6.

Encouragingly, clustering reduced the exit rates on all of the workloads. But, as the following calculation shows, these workloads are not dominated by exit costs. We assume a combined hardware and software cost of servicing an exit at 3000 cycles. Then, on the 3.33 GHz machine used here, each exit saved by clustering amounts to 0.9 us of CPU time. For the Mail Server workload, clustering saves about 1400 us of CPU time each second, i.e., just 0.14% of a core. So we see a small win, but one that is too small to detect on the bottom line.

While this result may at first seem disappointing, we note three positives: (1) clustering is either a net win or neutral but never a loss; (2) for some latency-sensitive workloads, including request-response client/server workloads and HPC workloads over a low-latency network, even a microsecond may matter; (3) we have found many actual workloads for which clustering delivers a bottom-line visible throughput improvement.

As a final dimension to the performance and effectiveness of clusters, let us relate them to work that has been done in the space of device virtualization. We ran netperf in a VM on VMware Workstation to determine if clusters can improve virtualized network performance. Netperf can measure either throughput or latency. We used the latter, configuring netperf to measure number of network packet roundtrips achievable per second. Meanwhile, in the hypervisor, we counted number of exits per second, allowing us to "normalize" exit counts per network roundtrip. With this setup, we first gave the VM a virtual e1000 NIC. This NIC is an emulated version of a well-known physical NIC produced by Intel. For e1000, with clusters disabled, each network roundtrip induces 2.6 exits. With clustering enabled, the exit count drops by 24% to 1.97 exits per roundtrip.

Replacing e1000 with a paravirtualized vmxnet NIC, a device specifically designed to be virtualization-friendly (i.e., designed to require fewer device touches from guest software) we found that clusters drop the exit rate per roundtrip from 1.3 to 1.2, a reduction of about 8%.

We are encouraged that clusters make a standard e1000 NIC perform measurably better and closer to the level of performance of a paravirtualized NIC. While the latter remains faster, extra steps are needed when switching from an e1000 NIC that works "out of the box" to a paravirtualized NIC that requires an add-on driver. This means that many virtual machines will continue to use an e1000 NIC.

We also find it interesting that a guest running with the paravirtualized NIC benefits nontrivially from clusters. Most likely, the measured improvement can be attributed to guest software outside of the NIC driver per se, perhaps on the general interrupt delivery path, although we have not been able to determine this with certainty.

10 Related Work

The cost of an exit has three components: the hardware transition where, typically, microcode switches the CPU from executing guest code to executing hypervisor code, the software transition where hypervisor code saves remaining guest state and loads hypervisor state, and the actual handling of the exiting guest instruction. Moreover, the first two cost components have counter-parts that apply when resuming guest execution. Clusters improve virtual machine performance by avoiding transition costs. They do not significantly speed up the actual handling of instructions that would, absent clustering, have caused their own exits.

Exits can be avoided using hardware or software techniques, or a combination thereof. The history of build-

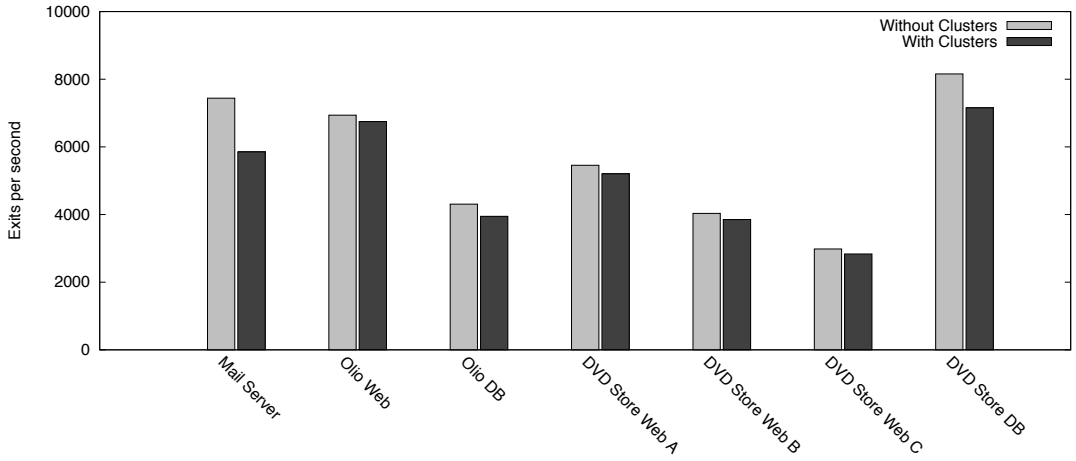


Figure 6: Exit rates for workloads from VMmark.

ing hardware that avoids certain types of exits goes back to the classical IBM mainframe era. Initial mainframe VMMs executed the guest with reduced privileges, causing all privileged instructions to trap. Soon, however, hardware support emerged to avoid many of these traps. For example, IBM's System 370 architecture introduced *interpretive execution* [12], a new hardware execution mode for running guest operating systems. This architectural extension specifies an encoding of guest privileged state and provides a new `SIE` "start interpretive execution" instruction to enable the VMM to hand off guest execution to the actual hardware. Many instructions that would otherwise have trapped in a simple deprivileged execution environment now can run directly in hardware against the provided guest privileged state.

Today, Intel and AMD use the same approach to avoid many virtualization exits. As previously explained, contemporary x86 virtual machines do not merely run in a less privileged protection ring, but run in guest mode with a VMCS (Intel terminology) or a VMCB (AMD terminology) capturing select vCPU state. Intel's `VMRESUME` and AMD's `VMRUN` instructions are equivalent to IBM's `SIE` instruction. As was the case on mainframes, this arrangement allows privileged x86 instructions like `CLI` (disable interrupts) and `STI` (enable interrupts) to execute without taking exits in guest mode: the instructions simply clear and set the vCPU's interrupt flag (`IF`) and check for a pending interrupt whenever `IF` changes from 0 to 1.

Along the same lines, and even more recently, both AMD and Intel have added hardware support for virtualizing the x86 MMU. This hardware support eliminates the need for the VMM to implement shadow page tables to virtualize guest virtual memory. Instead, the RVI/EPT hardware extensions allow the guest and hypervisor, each, to designate a set of page tables that map from guest virtual to guest physical memory and from guest physical memory to host physical memory, respectively. Giving the hardware MMU access to these two sets of page tables significantly reduces the number of exits required for running workloads that have frequent context switches, frequent page faults, and/or frequent memory map/unmap operations [7, 1]. However, these techniques come at the price of higher TLB miss costs and additional MMU implementation complexity.

Turning now to software techniques, paravirtualization [5] was initially used to overcome obstacles to virtualizing the x86 architecture without the need for deeper software techniques such as binary translation [1]. Specifically, if a paravirtualized guest kernel avoids all use of non-trapping privileged instructions such as `POPF`, it can run on a trap-and-emulate VMM. Additionally, paravirtualization was used to improve performance using a batched hypercall interface to amortize the guest/VMM crossing overhead. For example, a bulk update of shadow PTEs was deemed beneficial in the days prior to RVI/EPT hardware support for memory virtualization [4]. Batching of hypercalls resembles our clustering technique, but with the key difference that clusters are "native hardware compatible" whereas (batched) hypercalls only work on a hypervisor and possibly not even across different hypervisors.

Paravirtualization has also been used to accelerate virtual machine I/O. For example, instead of giving the guest operating system a virtual device that behaves exactly like an existing physical device, the VMM may implement a special-purpose NIC that has been developed specifically for the virtualized environment. Such a paravirtualized NIC side-steps the complexity/performance trade-off that a "real" NIC implemented in silicon must

obey. It can present an interface to the guest that allows packets to be sent using fewer device "touches" than would be the case for common physical NICs. Xen's device architecture, for example, uses a shared memory area to allow the guest's network driver to communicate asynchronously with the hypervisor [5]. VMware's SVGA graphics device also uses a shared memory FIFO to transmit graphics commands from guest to hypervisor in a low-overhead manner [8].

While device paravirtualization delivers performance gains, it requires that the hypervisor vendor provides not just a VMM but also guest drivers for the paravirtualized devices. Thus, device paravirtualization realizes performance benefits only in exchange for taking on the burden of writing both device emulation code and guest drivers, and probably for multiple guest operating systems. As shown in Section 9, clustering can deliver some of the same benefits as device paravirtualization by reducing the number of exits required to complete an I/O transaction against a standard device.

Pass-through of physical devices to virtual machines eliminates exits in a different manner. With pass-through (also sometimes called "direct device assignment"), guests contain drivers for physical devices, permitting exit-free programming of I/O requests. Gordon et al. [9] describe how a hypervisor can carefully manage physical resources, such as the Interrupt Descriptor Table and the APIC, to further eliminate most interrupt-related exits.

In the Turtles project, Ben-Yehuda et al. took the lead in investigating performance of nested virtualization on Intel x86 processors with VT-x [6]. They discuss virtualization of EPT using EPT shadowing, and discuss device performance extensively, including the pass-through scenario. Impressively, Turtles achieves nested virtual performance within 6–8% of non-nested virtual performance for nontrivial workloads. Much as in our own nested virtualization work, Turtles uses VMCS shadowing to multiplex ("flatten") the inner hypervisor's use of VT-x/EPT onto the underlying physical CPU.

However, as explained earlier, Intel's use of special VMCS accessor instructions, `VMREAD` and `VMWRITE`, presents a particular challenge to nested performance since use of these instructions in the inner hypervisor triggers exits. It appears that to get the best performance, the Turtles project paravirtualizes `VMREAD` and `VMWRITE`, allowing the inner hypervisor to instead use simple memory reads and writes. (This creates exit-behavior similar to what would have been the case on an AMD CPU where VMCB fields are memory mapped.) Since use of paravirtualized VMCS accessors prevents the inner hypervisor from working in a unnested setup, Ben-Yehuda et al. propose use of binary rewriting to convert inner hypervisor `VMREAD` and `VMWRITE` instructions into memory-accesses on the fly. Such code rewriting resembles the dynamic code rewriting techniques proposed by LeVasseur et al. in that they are guest visible [11]. In contrast, our clustering technology does not leave any visible traces that the inner hypervisor can detect, other than indirect timing effects. Moreover, clusters apply not just at the outer hypervisor level (when the inner hypervisor cooperates) but also allows the inner hypervisor to collapse multiple inner guest exits into faster-executing clusters.

11 Future Work and Conclusions

The exit avoidance techniques described in this paper have been incorporated in recent VMware ESX, Workstation, and Fusion products. They have been enhanced from release to release and have been running successfully on customer systems for a few years. While we feel that the work has probably reached a sweet spot for today's CPUs and workloads, several directions of future work exist that may take this system to the next level.

First, an adaptive cost/benefit model can more accurately estimate when translations are justified. This would include modeling the costs of expensive translations (e.g., gap-filler memory accesses) and the savings of avoided exits. To capture differences across CPUs or between native and nested executions, the calibration must be done dynamically, e.g., during VMM power-on.

Second, cluster formation can be generalized. While we currently support intra-cluster control-flow, translations are still fairly restrictive: they are small, consecutive, control-flow permits only conditional jumps, and clusters always begin with an exiting instruction. Removing these restrictions permits more powerful clusters such as: loop clusters where the initial exiting instruction is in the middle of the loop; clusters with non-adjacent instructions; and clusters that span call/return control instructions. A branch prediction mechanism could point the translator towards the more frequent execution paths.

Third, caching page walks can reduce the cost of gap-filler memory accesses. For each data memory reference in a cluster, we currently generate a separate walk of the guest page tables to translate the guest linear address to a guest physical address. Often, multiple references to the same guest linear page exist in the same cluster. A simple cache of page translations maintained for the duration of a cluster invocation could reduce the cost of page table walks, in turn permitting us to relax the constraint on the number of memory references in a cluster.

The hardware costs of a virtualization exit have generally improved from one CPU generation to the next. However, the rate of improvement has been slowing. As the hardware round-trip latency bottoms out, efforts to improve virtualization performance must shift to techniques for avoiding virtualization exits. For nested virtu-

alization, exit avoidance is even more important, because virtual hardware is way behind on the curve. Virtualization exits can either be avoided through more sophisticated hardware, or by software techniques such as those described in this paper.

Acknowledgments. We would like to thank all current and past members of the VMM team at VMware that contributed to this work with code, discussions, or code reviews. Joshua Schnee and Bruce Herndon helped us with running VMmark. Finally, we thank the reviewers and our colleagues, especially E. Christopher Lewis, for insightful comments on drafts of this paper.

References

[1] ADAMS, K., AND AGESEN, O. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (2006), pp. 2–13.

[2] AGESEN, O., GARTHWAITE, A., SHELDON, J., AND SUBRAHMANYAM, P. The evolution of an x86 virtual machine monitor. *SIGOPS Oper. Syst. Rev. 44*, 4 (Dec. 2010), 3–18.

[3] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, June 2010. Chapter 15.

[4] AMSDEN, Z., ARAI, D., HECHT, D., HOLLER, A., AND SUBRAHMANYAM, P. VMI: An interface for paravirtualization. *Ottawa Linux Symposium* (2006).

[5] 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 *SOSP '03: Proceedings of the nineteenth ACM symposium on operating systems principles* (2003), pp. 164–177.

[6] 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 *OSDI '10: 9th USENIX Symposium on Opearting Systems Design and Implementation* (2010), USENIX Association.

[7] BHARGAVA, R., SEREBRIN, B., SPADINI, F., AND MANNE, S. Accelerating two-dimensional page walks for virtualized systems. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems* (2008), pp. 26–35.

[8] DOWTY, M., AND SUGERMAN, J. GPU virtualization on VMware's hosted i/o architecture. *SIGOPS Oper. Syst. Rev. 43*, 3 (2009), 73–82.

[9] 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 seventeenth international conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS '12, ACM, pp. 411–422.

[10] INTEL CORPORATION. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2*, January 2011.

[11] LEVASSEUR, J., UHLIG, V., CHAPMAN, M., CHUBB, P., LESLIE, B., AND HEISER, G. Pre-virtualization: Slashing the cost of virtualization. Technical Report 2005-30, Fakultät für Informatik, Universität Karlsruhe (TH), Nov. 2005.

[12] OSISEK, D. L., JACKSON, K. M., AND GUM, P. H. ESA/390 interpretive-execution architecture, foundation for VM/ESA. *IBM Systems Journal 30*, 1 (1991), 34–51.

[13] POON, W.-C., AND MOK, A. K. Bounding the running time of interrupt and exception forwarding in recursive virtualization for the x86 architecture. Technical Report VMware-TR-2010-003, VMware, Inc., 3401 Hillview Avenue, Palo Alto, CA 94303, USA, Oct 2010.

[14] ROBIN, J. S., AND IRVINE, C. E. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2000), USENIX Association.

AppScope: Application Energy Metering Framework for Android Smartphones using Kernel Activity Monitoring

Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, Hojung Cha
Dept. of Computer Science,
Yonsei University, Korea
{cmyoon,dkim010,wwjung,ckkang,hjcha}@cs.yonsei.ac.kr

Abstract

Understanding the energy consumption of a smartphone application is a key area of interest for end users, as well as application and system software developers. Previous work has only been able to provide limited information concerning the energy consumption of individual applications because of limited access to underlying hardware and system software. The energy consumption of a smartphone application is, therefore, often estimated with low accuracy and granularity. In this paper, we propose AppScope, an Android-based energy metering system. This system monitors application's hardware usage at the kernel level and accurately estimates energy consumption. AppScope is implemented as a kernel module and uses an event-driven monitoring method that generates low overhead and provides high accuracy. The evaluation results indicate that AppScope accurately estimates the energy consumption of Android applications expending approximately 35mW and 2.1% in power consumption and CPU utilization overhead, respectively.

1 Introduction

With the widespread use of smartphone applications, the energy consumption of each application is important information that is used to manage a device's power. Smartphone users can adaptively select energy-efficient applications based on the energy consumption of an application. Additionally, understanding the energy consumption of each process or hardware component is a key area of interest for application and system software developers [1-5].

Estimating the energy consumption of a smartphone application is practically difficult. The estimation system should be able to determine the power usage of various hardware components in the device. However, this information is difficult to acquire because of complicated hardware schematics and compact form factor. An accurate power model for hardware components should be available to determine the level of energy consumption for each component.

Previous work has addressed various energy metering methods for smartphones [6-9]. However, these earlier models are lacking in terms of granularity and accuracy. An example of a previous system is PowerScope [6], which provided the energy consumption of applications at a fine-grained level, but required post-processing using an external device. The developer also needs to import related APIs for energy metering. PowerTutor [7, 8] proposed an estimation method for hardware components, but the system does not provide energy information for each application or process. PowerProf [9] required information on the API usage for each application in order to estimate energy consumption. The limitations of the above mentioned models are the result of schemes that focus on the accuracy of the power model but do not consider the actual usage of the hardware component. Accurate estimation of an application's energy consumption depends on the accuracy of the power model and the accuracy of a hardware component's usage statistics. In particular, the hardware usage estimation is critical because it is a pre-requisite for estimating the energy consumption of smartphone applications.

Usage estimation for hardware components has been previously completed using hardware performance counter (HPC) [5, 10-13], software performance counter (SPC) such as the Linux *procfs/sysfs* [7, 14-16], or *BatteryStats*, which is provided by Android [8, 17]. However, depending on the process or underlying hardware components, these approaches provide different information. Thus, obtaining accurate usage statistics for each hardware component is limited by this feature.

In this paper, we propose a software scheme, called AppScope. This scheme automatically estimates the energy consumption of applications running on Android smartphones. The proposed system accurately estimates the usage (or utilization) statistics for each device component. We have designed the scheme based on monitoring the Android kernel at a microscopic level. In order to estimate the usage statistics of each application, the system analyzes the traces of a system call, as well as the messages for Android binder inter-process communication (IPC). AppScope collects usage information based on an event-driven approach; hence, the energy consumption of each application is estimated at a fine-grained level. Additionally, the proposed approach is

applicable for any Android-based device, without modification of system software, because we implemented the scheme using a dynamic module in the Linux kernel.

The contributions of our work are as follows:

- AppScope provides the energy consumption of Android applications automatically, being customized to the underlying system software and the hardware components in the device.
- AppScope accurately estimates, in real-time, the usage of hardware components at a microscopic level.
- We implemented AppScope as a loadable kernel module to improve portability of the proposed approach. Thus, AppScope can be used on an Android-based device without modifying the system software.

2 Backgrounds

The accuracy of application energy metering and granularity of measurement depend on power and energy models. In this section, we discuss the models and briefly discuss DevScope, which provides a nonintrusive, online power analysis of smartphone hardware components.

2.1 Power and Energy Models

Depending on the power interdependencies among underlying hardware components, power models are typically classified into a linear or a non-linear regression model. The non-linear models often capture power dependency among hardware components, although their performance does not significantly outperform linear models [18]. We only consider linear models in this paper.

With a linear model, the power consumption P of a device is expressed as follows:

$$P = \Sigma_i(\beta_i \times x_i) + P_{base} + P_\epsilon \quad (1),$$

Here, x_i represents the vector of usage measurement for hardware component i and β_i the power coefficient for component i. Also, P_{base} is the base power consumption, and P_ϵ is a noise term that cannot be estimated by the model. Then, the total energy consumption E of a smartphone is expressed as:

$$E = \sum_j E^j + (P_{base} + P_\epsilon) \cdot D, \text{ where}$$
$$E^j = \sum_i(\beta_i \times x_i^j) \cdot d_i^j \quad (2)$$

Here, D is the device's power-up duration. E^j is the energy consumed by process j. E^j is expressed with β_i, x_i^j, and d_i^j, where x_i^j and d_i^j represent the usage vector and active duration of hardware component i accessed by process j, respectively. Note that the accuracy of E^j is influenced by β_i, x_i^j, and d_i^j. To estimate the energy consumption of smartphone applications, it is essential to obtain accurate values of β, x, and d in an effective way. Note that AppScope employs a linear model to estimate the energy consumption of smartphones.

2.2 DevScope

Previous studies on linear power modeling for mobile devices [7, 14, 15] used external power measurement to profile β_i for each device type. In practice, β_i varies, even on the same type of device, depending on hardware, software configuration, and battery status [16, 19]. Typically, these values are directly obtained with hardware measurements for target devices; hence, this offline method is costly and hardly adaptive to changing environments.

The limitation of the offline method can possibly be overcome by using an online approach that employs a battery monitoring unit (BMU) [16, 19], which is built in to smartphones. The scheme would enable the implementation of an online power model that automatically constructs a power model for each device, adapting to changes of external factors, such as aging or software updates. However, in order to employ a BMU as an online power measurement tool, we must consider two factors that are inherent in the properties of BMU. First, the information update rate of a BMU is noticeably lower than external measurement tools; hence the online results may not be accurate. Second, since the user is not able to intervene in the process of constructing a model, it is difficult to understand the exact relationship between system activities and power consumption.

DevScope [19], an Android application, is an automatic and online tool used to generate a power model for smartphones. The tool probes operating systems to obtain information about individual component types and their configurations. Additionally, by monitoring the update activity of BMU, DevScope detects the update rate automatically. According to individual component types, system configuration, and BMU update rates, DevScope dynamically creates a control scenario for each hardware component to perform power analysis. Hence, even though a device (i.e., smartphone) is identical, the scenario might be different due to each device's configuration. The scenario assigns a workload to each component, which then triggers every possible power state of the component; for example, specific operations for CPU, display brightness, GPS on/off, and packet transmission for cellular and WiFi. Each workload is

maintained for a time period to collect enough measurement samples (i.e., 5 samples) to overcome the limitation of BMU's low update rate. DevScope turns off every other component, except for the component under measurement. However, since the CPU should be alive to measure the power consumption of other component, CPU power analysis is conducted ahead of other hardware components. The power analysis of other hardware components is then conducted by subtracting the power consumption of the CPU from the total power consumption of the device that is being measured by the BMU. While performing the test scenario, DevScope classifies the results into each term of the power model and then generates corresponding power coefficients. DevScope requires a user's explicit interaction to initiate training and collect power coefficients. Hence, if re-training is required due to changes in a system's configuration, the process should be repeated manually, yet the power coefficients will be updated automatically. Note that the training time depends on the characteristics of underlying hardware components, as well as the update rate of BMU. The process typically takes minutes.

Currently, DevScope uses the device power model, illustrated in Table 1, for five core hardware components of smartphones, that is CPU, display, cellular (3G), WiFi, and GPS, and generates their power coefficients β_i. To analyze the CPU characteristics, DevScope locates the frequency-voltage table using */sysfs*; thus the number of available frequencies is dynamically determined. The power consumption is then measured by setting the frequency to every value.

In the case of display, DevScope presently only supports LCD displays, not more modern display types, such as OLED. The tool dynamically generates a table that contains coefficients for every possible brightness level. This is because the relationship between power consumption and brightness level is not completely linear [15].

To determine the coefficient for cellular, DevScope considers power consumption of each RRC (radio resource control) state; IDLE, FACH, and DCH (see Section 4.5 for further details). The power state transition is proven via a planned scenario in which data traffic is controlled. The power consumption pattern of WiFi differs depending on the specific packet rate (i.e., threshold workload size) [7]. DevScope gradually increases the packet rate and finds the threshold value at which the power consumption pattern is changed. DevScope currently uses a fixed-strength signal model for both WiFi and 3G; hence, although the model would suit the purpose of application and system developers, the tool should be supplemented to reflect true mobile environments.

The power states of GPS are defined into three states:

Table 1: Power model for smartphone components

Component	Model
CPU	$P^{CPU} = \beta_{freq}^{CPU} \times u + \beta_{freq}^{idle}$ u: utilization, $0 \le u \le 100$ $freq$: frequency index, $freq = 0,1,2 \cdots, n$
LCD	$P^{LCD} = \beta_b^{LCD}$ b: brightness level, $\mathrm{MIN}(level) \le b \le \mathrm{MAX}(level)$
WiFi	$P^{WIFI} = \begin{cases} \beta_l^{WIFI} \times p + \beta_l^{base}, & if\ p \le t \\ \beta_h^{WIFI} \times p + \beta_h^{base}, & if\ p > t \end{cases}$ p: packet rate, t: threshold
cellular(3G)	$P^{3G} = \begin{cases} \beta_{IDLE}^{3G}, & if\ RRC\ state\ is\ IDLE \\ \beta_{FACH}^{3G}, & if\ RRC\ \text{state}\ is\ FACH \\ \beta_{DCH}^{3G}, & if\ RRC\ state\ is\ DCH \end{cases}$
GPS	$P^{GPS} = \beta_{on}^{GPS},\ if\ GPS\ is\ on$

OFF, SLEEP, and ACTIVE. Since the switch between SLEEP and ACTIVE states has a constant pattern, we regard SLEEP and ACTIVE states as ON.

The goal of AppScope is to provide a practical application energy metering system that is readily runnable on Android smartphone. AppScope estimates energy consumption E^j of each process based on a linear model, as shown in Equation (2). AppScope employs DevScope's component power model (see Table 1) and the power coefficient $\beta_{i,}$ which are obtained for target devices. Therefore, in this paper we focus on the AppScope features that deal with the automatic acquisition of x_i^j and d_i^j for each hardware component accessed by an application.

3 AppScope: The Application Energy Metering System

AppScope is an application energy metering framework for the Android system that uses hardware power models and usage statistics for each hardware component. AppScope provides accurate and detailed information on the energy consumption of applications by monitoring kernel activities for hardware component requests.

Figure 1 shows an overview of AppScope. The system conducts application energy metering via three phases:

(1) Detection of process requests that are accessing hardware components.

(2) Analysis of usage statistics and status changes of the requested hardware components.

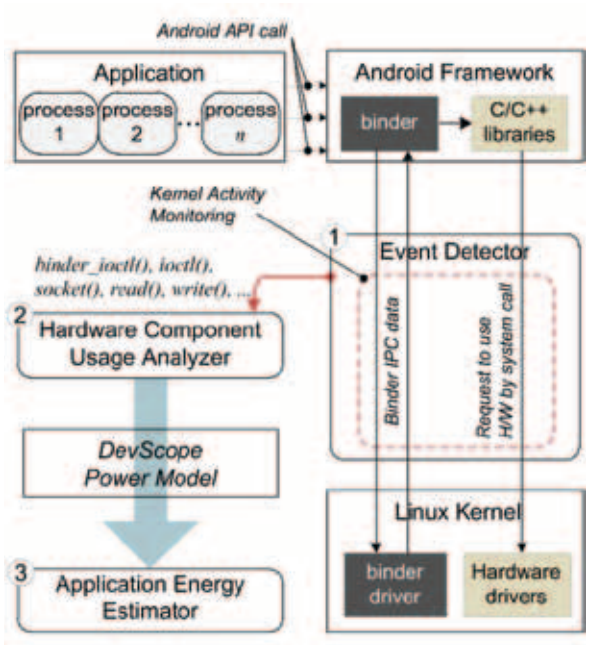


Figure 1: AppScope overview on Android platform.

(3) Linear model-based application energy estimation by adding up energy consumption of each hardware components accessed by application.

3.1 Event Detector

Event Detector probes system calls that are relevant to the hardware component operation, such as CPU frequency switching, process switching, packet transmission, and binder I/O control. Event Detector monitors *cpufreq_cpu_put()* for CPU frequency switching, and *sched_switch()* for process switching. For packet transmission operations, the usage of the *dev_queue_xmit()* and *netif_rx()* kernel functions are monitored. For binder I/O control, Event Detector monitors *binder_transaction()* which is a part of *binder_ioctl()* routine. These detections are passed onto the Hardware Component Usage Analyzer.

3.2 Hardware Component Usage Analyzer

When an event is detected, the Hardware Component Usage Analyzer collects usage statistics for each hardware component and data that is required to apply the power model to the component. Each hardware component is activated by different kernel operations. Moreover, the type of information required to apply the power model varies depending on the characteristics of power consumption. Therefore, the method of collecting information is separately defined according to the hardware components. Figure 2 illustrates different methods for the components. In the case of the CPU, the

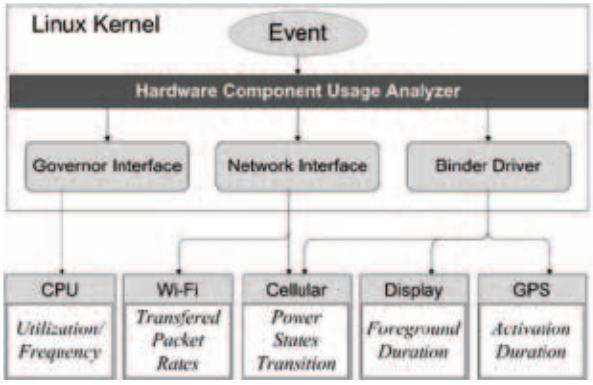


Figure 2: Hardware Component Usage Analyzer

changes in utilization and frequency are collected by referring to the governor interface. For WiFi, the rate of the transmitted/received packets of a process is collected by monitoring the data flow in the Linux networking stack. In the case of LCD display and GPS, the duration of activation is investigated by analyzing the IPC interfacing message of the Android binder. For 3G interface, the information on transmitted/received packets and the changes in power state are collected through the Linux networking interface and the Android IPC binder interface, respectively. The detailed process for each hardware component is presented in Section 4.

3.3 Application Energy Estimator

The energy consumption of an Android process is estimated via hardware usage statistics, which are applied to the underlying power model for hardware components (see Table 1). The application energy consumption is then obtained by combining the energy consumption of all processes that belong to an application. In the Android platform, each application has a unique user id (UID) to prevent other applications from accessing its specific resources. AppScope differentiates the energy consumption of an application using UID.

In our work, we assume that the overall energy consumption of a device running an application includes both "system energy" and "application energy". System energy is defined as a basic consumption that is required to operate a device using the Android framework. It includes the energy consumption for various Android system processes as well as for the Linux kernel threads. Meanwhile, application energy is defined as consumption solely used by the processes belonging to an application. In terms of UID in the current Android framework, UID=0 is used by the root-owned processes, the UIDs around 1,000 are used by the Android system processes, and the UIDs over 10,000 are used by applications. AppScope estimates both application energy and system energy consumption.

4 Application's Hardware Usage Analysis

In this section, we describe AppScope's techniques that are used to detect and analyze how each hardware component is used by an application.

4.1 Limitation of Previous Approaches

Conventional methods for estimating hardware component usage include HPCs, *procfs* and *sysfs* on Linux, and *BatteryStats* on Android. Each of these methods is limited in terms of their efficiency in application energy metering.

HPCs are a set of special registers that are built into microprocessors and are used to count certain processor events. These counters can be used for low-level performance evaluations or system tuning. With the use of HPCs, power consumption can be accurately analyzed. However, HPCs are highly dependent on a processor's architecture, and kernel modification is generally required to look into the HPC registers. Moreover, the counting results are effective only for CPU-and memory-related power analysis.

The Linux *procfs/sysfs* are special filesystems in Linux that provide information about processes, hardware usage, and other types of system information; *procfs/sysfs* are inadequate for monitoring application energy. First, the update rate of each hardware component is different, as is the data access method. For instance, with the Linux kernel 2.6.35.7 for Android Gingerbread, the update rate of CPU utilization is 5Hz and the CPU frequency is provided only for the current status. It is therefore difficult to decompose the CPU utilization of an application into each frequency. Also, due to the constraints in *procfs/sysfs* access, the application energy metering system should continuously poll both CPU utilization and frequency status to estimate CPU energy consumption. Second, the details of the information obtained from the filesystem vary depending on the type of underlying hardware. For example, WiFi traffic is not provided for process bases and GPS usage information is generally not available. Last, although the aforementioned limiting factors can be alleviated with kernel modification, the kernel should generally not be modified to support system portability on diverse platforms.

The Android *BatteryStats*, which provides battery status and hardware usage information, is a widely-used functionality for battery-related applications. *BatteryStats* inherits the fundamental limitations of *procfs/sysfs* and per-process usage information is not available for a certain type of hardware component. Furthermore, the granularity of information varies with hardware components. For example, *BatteryStats* produces component usage statistics on CPU and WNI traffic by reading *procfs/sysfs*, whereas display utilization is only available for the entire system.

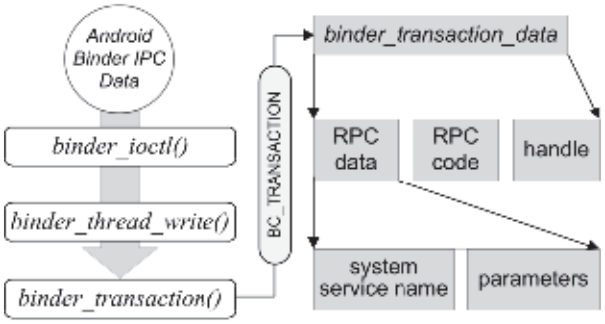


Figure 3: The Android IPC Data format for RPC procedure.

4.2 Kernel Activity Monitoring

Android applications typically access hardware components in two different ways. When an application uses hardware components supported by the Linux kernel, the application requests related system calls. Otherwise, application requests RPC via the Android binder [20, 21]. This section explains how AppScope uses the Android binder RPC mechanism to analyze component usage, and also how usage data are collected upon system calls.

4.2.1 Android Binder RPC

Android RPC is executed using binder RPC protocol, which is processed in the binder driver of the kernel. Figure 3 shows the data format of the Android IPC that is used for processing the BC_TRANSACTION command of the binder RPC procedure. To execute the stub interface of many service applications, the BC_TRANSACTION command is sent to the binder driver. At this moment, IPC data is sent to the binder driver with *binder_ioctl(),* and *binder_transaction()* executes the BC_TRANSACTION command within the binder driver. Thus, AppScope analyzes IPC data processed in *binder_transaction()* and collects data about the system usage. BC_TRANSACTION differentiates the requested functions using the RPC code of *binder_transaction_data*, as shown in Figure 3. The details of the requested command are known as "System Service Name" and "Function Input Parameter" within the RPC data.

4.2.2 Kprobes

Kprobes [22] is used to monitor the behavior of system calls. Kprobes is one of Linux's debugging mechanisms. It can dynamically insert break points during a kernel's runtime. It can be inserted into any kernel routine and

collect information non-destructively and without intruding into original kernel behavior. With this mechanism, the kernel function call can be monitored with low overhead because only a single instruction is substituted to detect the kernel operation. AppScope uses Kprobes to detect events on hardware component operations and to analyze a component's usage statistics. AppScope is compiled as a kernel module and controlled dynamically. Hence, apart from installing and removing the module, no additional user activity is required.

4.3 CPU Usage

In order to measure the consumed energy of process P_x, we need utilization u, as well as the CPU frequency relevant to u for a given time unit. In the Linux kernel, the utilization of P_x is computed using P_x's *utime()/stime()*. The *utime()/stime()* is estimated by detecting the switch from the TASK_RUNNING state to another one. Here, checking the states of all processes and updating their utilization for each scheduler call would generate a significant overhead. To reduce the overhead, AppScope detects the process switch by monitoring a wake-up event via *sched_switch()*. When a wake-up event occurs, AppScope updates the utilization of the previous process to calculate the utilization.

The CPU frequency changes according to the dynamic voltage and frequency scaling (DVFS) governor in the kernel. The *cpufreq_cpu_put()* function invokes a change in the frequency of the DVFS governor. Thus, the function is monitored and the frequency information is obtained at the call time. Frequency information, as well as information regarding system time, is then stored. Figure 4 illustrates the concept of the mechanism. Here, both the frequency change and the utilization value are computed based on the system time (jiffies), and each color indicates a separate process.

4.4 WiFi Usage

The energy consumption of WiFi varies according to the packet rate (i.e., transmitted packets per second). Thus, the amount of transmitted WiFi packets per given unit should be estimated to compute the energy consumption of process P_x. The data packet rate of process P_x depends not only on data size but also on protocol and maximum transmission unit (MTU). In our system, we have referred to the device agnostic network interface (DAI) layer of the Linux networking stack to estimate the packet rate. The DAI layer is an abstract layer located directly above the device driver layer (DDL), and it prepares (independently from the protocols) data for eventual transmission. In DAI, there are two main functions: *dev_queue_xmit()* for transmitted data and *netif_rx()* for received data. Figure 4 shows the WiFi

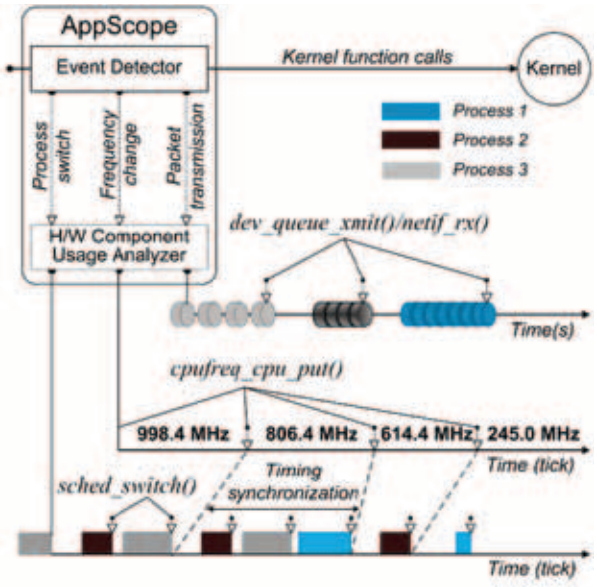


Figure 4: Analysis of CPU utilization/frequency, and the WiFi interface.

usage analysis of a process based on the detection of *dev_queue_xmit()* and *netif_rx()* calls. The packet rate is computed using the transmission/reception time of the packet. The power state is then identified based on the packet rate, and energy consumption is computed using activated time duration.

4.5 3G Usage

The energy consumption of a 3G interface depends on the RRC state. To efficiently utilize a radio resource in a 3G network, the RRC protocol typically defines three states: IDLE, FACH (forward access channel), and DCH (dedicated channel). Although the RRC state change depends on a carrier's policy, the RRC, in general, remains in the IDLE state when there is no data to send or receive. The state switches to the low power state, FACH, when data communication starts, and remains in the high power state, DCH, while data is being sent or received [23]. Our work is conducted in the Korea's SK-Telecom WCDMA network. In this network, mobile phones remain in the IDLE state if there is no data transmission. When data communication occurs, the mobile phone connects to the UMTS network for a short period of time, and then accesses the HSDPA network for a high-speed data transmission accompanying the RRC state transition. Thus, we identify the state transition of RRC based on the connection type of network.

The radio interface layer (RIL) daemon and vendor RIL of the Android telephony service are both located in the Linux user space. That is, voice calls and control commands are not processed using the Linux networking stack. Hence, 3G usage and the RRC state transition,

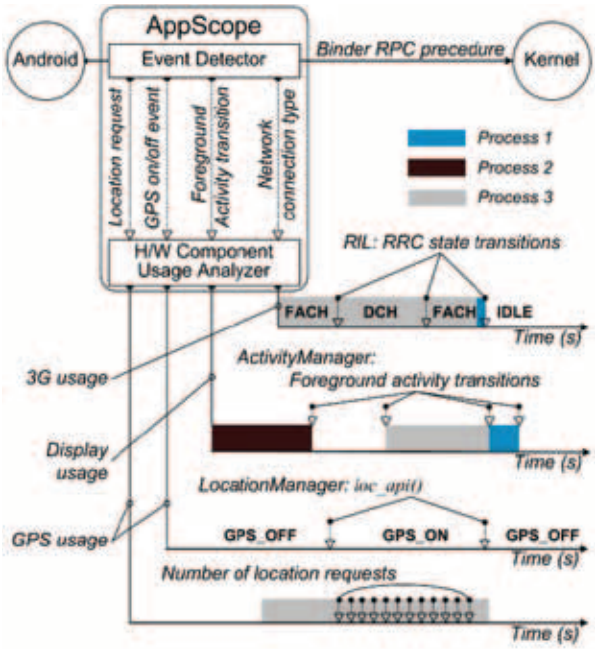


Figure 5: Analysis methods for 3G, GPS interface and LCD display usage information.

with the exception of data communication, cannot be analyzed within the Linux kernel. We therefore analyze the hardware component operation using the Android binder RPC. Figure 5 illustrates the concept of AppScope regarding 3G usage statistics. The change in network connection type is detected by checking the IPC data in Android RIL.

4.6 LCD Usage

The energy consumption of an LCD display is proportional to display brightness and display duration. Brightness can easily be identified from the current display settings of the Android framework. However, display usage, per application, cannot be directly obtained using the device routine within the kernel because the display operation is controlled by the Android framework. Therefore, AppScope recognizes foreground applications using the Android *ActivityManager* service, and its display usage is estimated by monitoring it. AppScope catches an event on foreground activity by checking the IPC data in the binder driver. When the process P_x's activity is in the foreground, display usage data is updated until another activity is brought into the foreground or the screen is turned off.

4.7 GPS Usage

The energy consumption of GPS is directly related to the power-on time of the interface. However, on/off time of a GPS system does not depend on the location request of the application. Also, several applications may simultaneously request location information from a GPS interface. Since the device interface for GPS is not exposed in the kernel, the estimation of process P_x's GPS usage is not trivial. In our work, we estimated cess P_x's usage statistics by monitoring *loc_api()* and *LocationManager* in the binder driver. The GPS interface is turned on/off with the *loc_api()*, and *LocationManager* provides location updates when GPS is turned on. Figure 5 illustrates how the AppScope estimates GPS usage of process P_x through monitoring the *LocationManager* of the Android framework. AppScope monitors *LocationManager* calls and calculates the GPS activation duration. During GPS activation, AppScope counts the location requests to *LocationManager*. The count is then used to estimate the energy consumption for each application process. Thus, when multiple processes request location information, AppScope distributes the energy consumption proportionally to the corresponding processes based on the usage count.

5 Evaluation

AppScope was developed in Linux kernel 2.6.35.7. The SystemTap version 1.3 [24] also uses Kprobes and data collection for the purpose of evaluation. All evaluations are carried out on HTC Google Nexus One (N1; Qualcomm QSD 8250 Snapdragon 1GHz, 3.7-inch Super LCD display) [25] with Android platform version 2.3. Note that N1 is equipped with a current sensor (MAXIM DS2784) upon which DevScope can build its power model. The Monsoon Power Monitor [26] is used as an external power meter.

In order to evaluate the AppScope framework, we

Table 2: Operation sequence of test applications

Time (sec)	Test App.	Operation	Description
0	Master	Run	Execution as a foreground activity and prevent screen off
20	Slave1	Transmit 2,000 packets via WiFi interface for 20 seconds	Approximate packet rates is 100pps
80	Slave2	Change the foreground activity for 20 seconds	After 20 seconds, Master app's activity return to foreground
120	Slave3	Start CPU job for 20 seconds	CPU frequency is changed by DVFS
160	Slave4	Transmit data via 3G interface for 20 seconds	RRC transition in the beginning of transmitting
200	Slave5	Turn on GPS interface for 20 seconds	Periodic updates of location information GPS

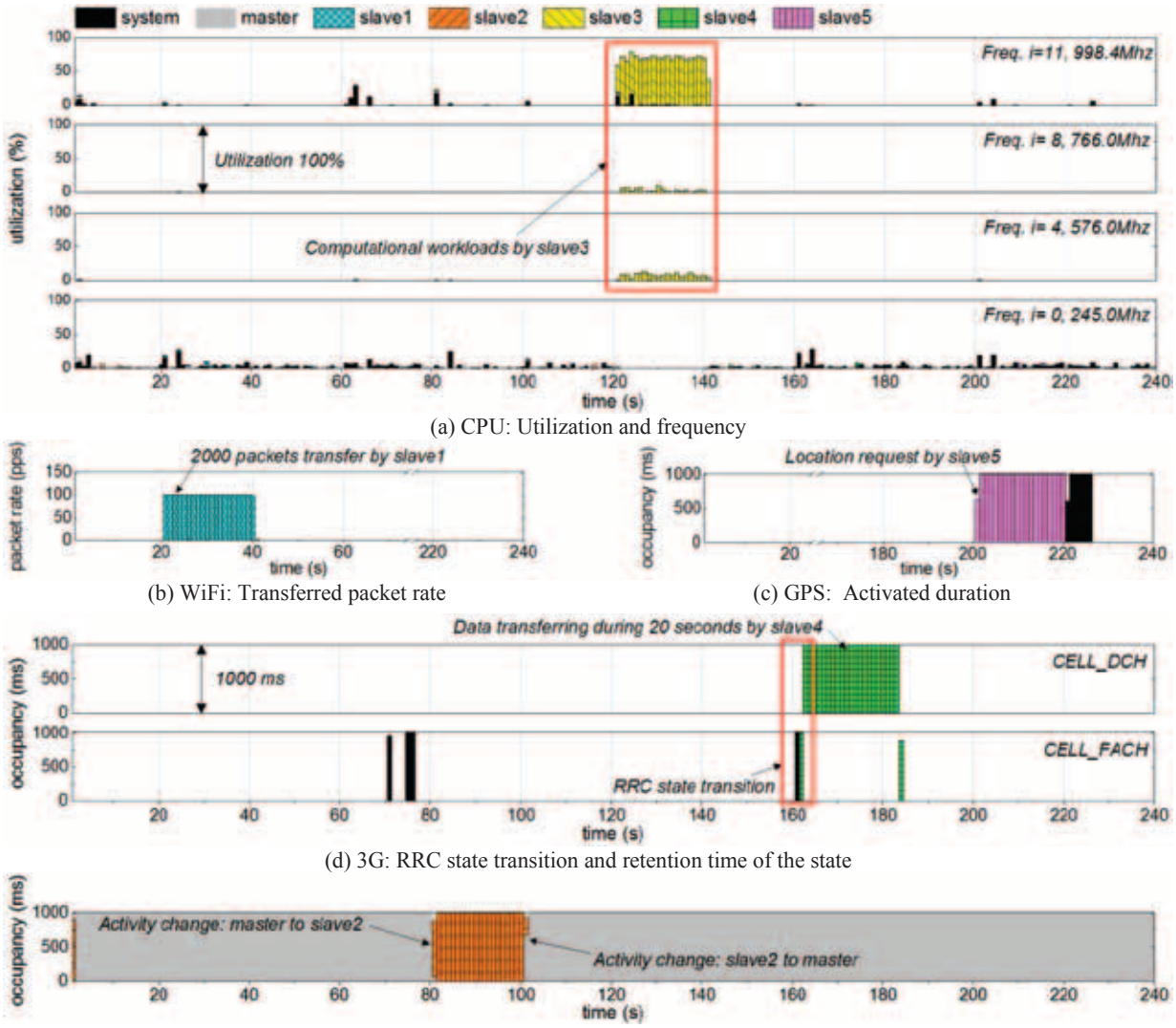


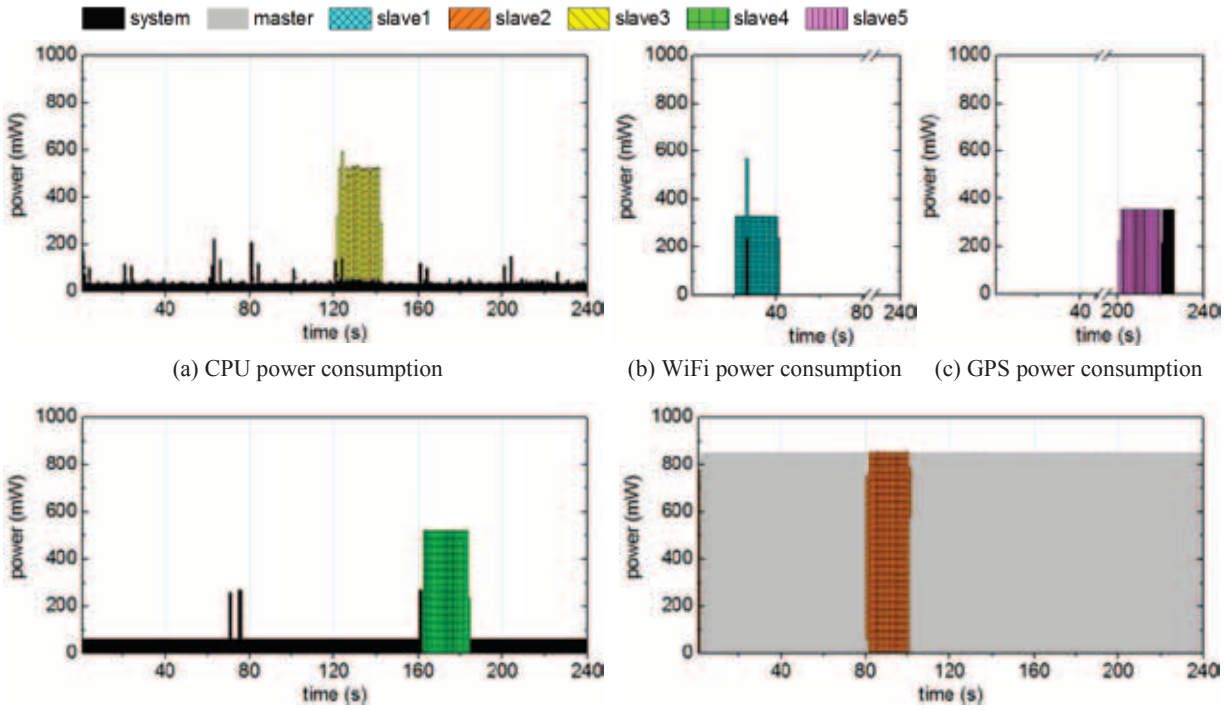
Figure 6: Hardware component usage trace of AppScope for test applications.

benchmarked a set of Android applications and estimated their energy consumption with AppScope. We also measured the overhead of AppScope in terms of power consumption and CPU utilization.

5.1 Component Usage Monitoring

To evaluate the accuracy of hardware event detection and collection of usage statistics, we designed and experimented on one "Master" and five "Slave" applications. The Master sets a pre-defined workload, executes the schedule of each hardware component workload, and controls the Slaves according to this schedule. We ran the Master and Slaves for 240 seconds in the order shown in Table 2.

Figure 6 shows the results of the tests on hardware component usage while running the test scenario in Table 3, where data was collected for every second. Each row in Figure 6(a) is differentiated by CPU frequency and *i* is the index in the frequency table for N1. The bar height represents utilization of relevant frequency. Due to space limitations in Figure 6(a), we have omitted some plots in which the utilization is too low or absent altogether. In the cases of GPS, LCD, and 3G, the power model requires activated time duration as usage information. The bar height in Figure 6 (c), (d), and (e) represents occupancy time (ms) in a unit time. The "system" stands for the system energy component, described in Section 3.3. The applications were started up at booting time and are differentiated by color.



(a) CPU power consumption (b) WiFi power consumption (c) GPS power consumption

(d) 3G interface power consumption (e) LCD display power consumption

Figure 7: AppScope power traces for test applications.

In Figure 6(a), from time *t*=121 for 20 seconds, Slave 3 actively utilizes the CPU. As a result, the CPU frequency increases, and the applications, which have been running in low frequency with low CPU utilization, started to operate in high frequency. Moreover, Slave 3 shows low CPU utilization in the same region with medium frequency due to the *OnDemand* policy.

In Figure 6(b), from time *t*=21 for 20 seconds, Slave 4 transmits data over WiFi. As shown in Table 2, Slave 1 sent 2,000 packets, i.e., its packet rate is 100pps. In Figure 6(c), Slave 5 uses GPS from *t*=201 for about 20 seconds. Note that after Slave 3 terminates the use of the GPS, the GPS is still used for about 4.5 seconds by the "system". With further experiments, we found that after an application terminates GPS usage, the "system" uses the GPS for a duration of about 2–4.5 seconds. After this timeframe, usage of the GPS interface is completely stopped.

Figure 6(d) shows that the "system" transmits some data before Slave 4 transmits data at time *t*=161. After that, the RRC state remains in FACH for a short duration and changes to DCH. Also, as the packet transmission is terminated, the RRC state changes to FACH. This result is consistent with the RRC protocol between carrier and mobile devices on UMTS networks. Figure 6(e) shows the switching point for display between the two foreground activities. When Slave 2 activity is brought to the foreground, the display is not used by any of the applications for a duration of about 60–100ms. This is a blank duration when the activity change occurs in the *ActivityManager*.

In summary, AppScope detects hardware operation time as indicated in Table 2. The experimental results show that AppScope observes accurate usage of hardware components and correctly observes their power characteristics.

Table 3: Power coefficient values of N1

Comp.	Index	Coefficient		Comp.	Index	Coefficient	
CPU	$freq$ *(Mhz)*	β_i^{freq}	β_i^{idle}	LCD	b	$\beta_b^{brightness}$	
	245.0	201.0	35.1		5	367.8	
	384.0	257.2	39.5		55	451.5	
	460.8	286.0	35.2		105	631.1	
	499.2	303.7	36.5		155	697.9	
	576.0	332.7	39.5		205	775.4	
	614.4	356.3	38.5		255	854.0	
	652.8	378.4	36.7	3G	rrc	β^{rrc}	
	691.2	400.3	39.6		IDLE	63.9	
	768.0	443.4	40.2		FACH	267.9	
	806.4	470.7	38.4		DCH	519.3	
	844.8	493.1	43.5	WiFi		β_l	β_h
	998.4	559.5	45.6		Transmit	1.2	0.8
GPS		β^{gps}			Base	238.7	247.0
	ON	354.7			Threshold	25pps	

5.2 Energy Metering Validation

We estimate energy consumption for each application based on hardware component usage, as shown in Figure 6. In order to attain an accurate estimation, we used DevScope [19] to extract power coefficients (Table 3), based on the power model explained in Table 1 of Section 2.2. Note that all the experiments for communication interfaces, such as WiFi, 3G, and GPS, were conducted at a stationary place, i.e., fixed-strength radio signals; hence, we did not consider energy effects on varying signal strength for these components. We compared the estimation results with the results obtained from the Monsoon power meter.

5.2.1 Granularity

Figure 7 shows the power consumption of hardware components per application. Figure 7(a) shows the CPU power consumption for the entire duration – 240 seconds. Overall, the "system" uniformly consumed approximately 100mW. The power consumption of Slave 3 is about 480 mW in the increased frequency region. As shown in Figure 7(b), (c), and (d), when communication components, such as WiFi, 3G, and GPS are used, we observed that the "system" consumes a certain amount of power. In Figure 7(e), when the application uses an LCD display, the power consumption of the LCD is relatively higher than other components. Master consumed the highest energy due to long display occupancy. However, it did not operate other hardware components. Table 4 shows the estimated energy results by aggregating the results shown in Figure 7. As shown in Table 4, AppScope provides application-specific energy consumption data for each hardware component, even when multiple applications run in parallel.

5.2.2 Accuracy

To analyze the correctness of energy consumption results obtained in Section 5.2.1, we compared our results with those obtained using the Monsoon power meter. Figure 8 shows the comparative results between AppScope estimation and external measurement. The

Table 4: Energy estimation of test applications

App.	CPU(J)	WiFi (J)	GPS (J)	3G(J)	LCD(J)	Total(J)
System	11.3	0.2	2.0	14.7	0	28.2
Master	0.5	0	0	0	186.8	187.3
Slave 1	0.04	6.80	0	0	0	6.84
Slave 2	0.1	0	0	0	17.9	18.0
Slave 3	9.3	0	0	0	0	9.3
Slave 4	0.01	0	0	11.41	0	11.42
Slave 5	0.01	0	7.00	0	0	7.01

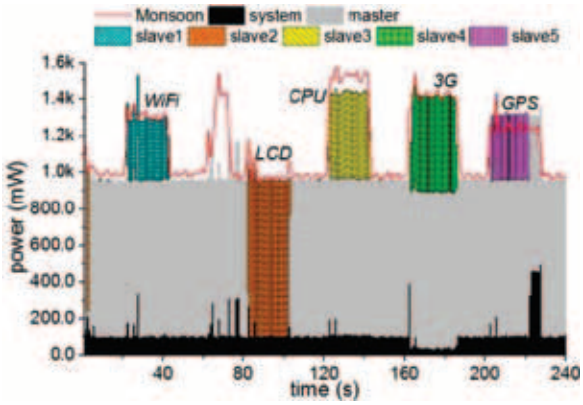


Figure 8: AppScope results vs. Monsoon measurement results for test applications.

aggregated power consumption of all applications using AppScope is similar to the entire power consumption measured using the external power meter. However, a power difference of about 100–400mW has been observed in some regions. At time t=60 for 10 seconds, the external measurement showed that power consumption temporally increased for a short period of time. This is because Slave 4 turned off the WiFi interface and the "system" automatically activated the 3G interface. In this process, AppScope noticed that the "system" sent packets over the 3G interface, but the 3G interface's power consumption was not detected due to the WiFi's turn off delay and 3G interface activation. When the CPU frequency rises, a large difference exists between the external measurement result and the power consumption estimated by AppScope. At time t=120 for 20 seconds, power consumption increases due to the CPU frequency and increased utilization. At this moment, the power consumption was estimated as 1400mW, which is 7% less than the external measurement result. This demonstrates the limitations of our simple CPU power model, which ignores the effects of cache, bus, memory and other SoC components. More accurate models can be built by using performance counters to account for these effects [5, 10-13]. Figure 8 summarizes that the overall energy consumption estimated by Monsoon is 282.8J, and 268.0J by AppScope, which is a 14.8J (5.2%) difference.

5.3 Overhead Analysis

To estimate the overhead of AppScope, we have performed the experiment described in Section 5.1 by loading and unloading AppScope onto the system. In both scenarios, power consumption is estimated using the Monsoon power meter. Figure 9 shows the results. During the experiments, test applications occupied the display activity. Therefore, the information regarding

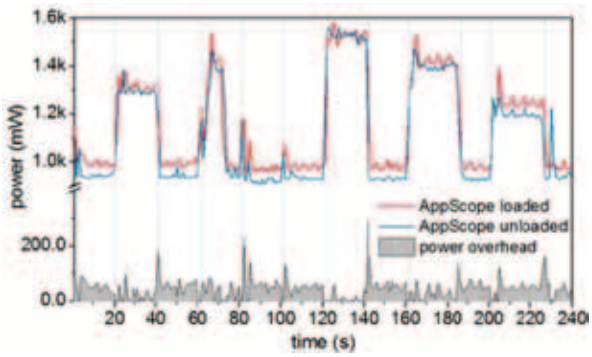


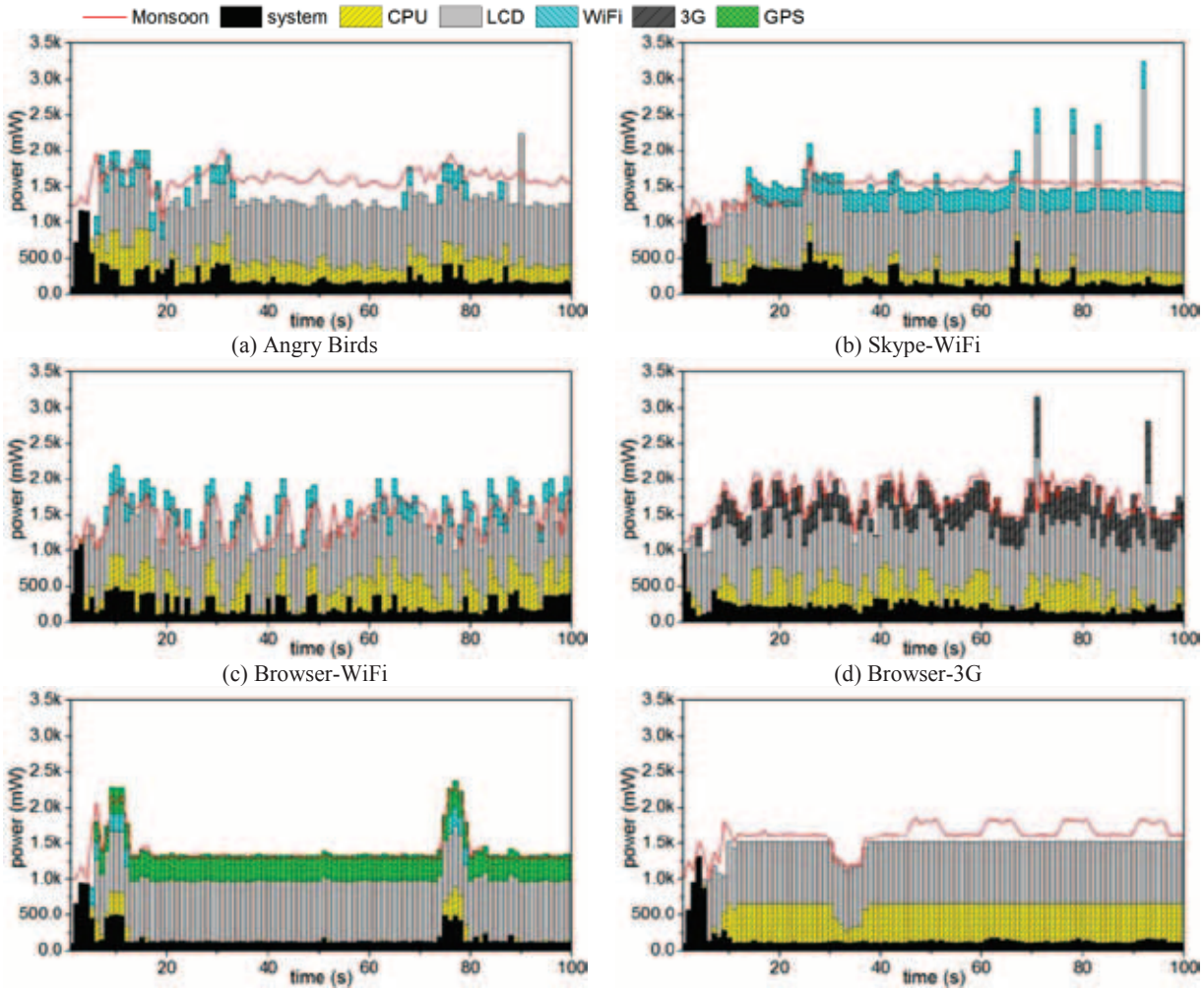
Figure 9: Overhead analysis of AppScope.

the power consumption of displays (Figure 9) was collected during the entire duration of the experiment. During CPU testing, the power consumption does not increase between 120-second to 140-second (see Table 2). While WiFi and 3G tests are carried out, the energy consumption slightly increases in comparison to the energy consumption experienced with the display only function. Within 240 seconds, AppScope generated 8.4J energy overhead, which is a 34.9mW increase on average. Moreover, the five tests showed that AppScope generated 2.1% CPU overhead on average, with a standard deviation of 1.9 and the worst case being 5.9%.

AppScope is a Linux kernel module and can be dynamically loaded/unloaded at runtime. Thus, users may install AppScope when analysis is required and remove it if unnecessary. Consequently, when AppScope is not activated, the overhead is not generated at all.

6 Real Application Energy Metering

We have evaluated AppScope's energy metering performance using applications distributed via *Google Android Market*. For this analysis, we have selected four applications that adequately utilize each component.



(a) Angry Birds
(b) Skype-WiFi
(c) Browser-WiFi
(d) Browser-3G
(e) Google Maps
(f) StabilityTest Benchmark

Figure 10: AppScope power traces for real applications.

Figure 10 shows the estimated energy consumption of Angry Birds (game), Skype (VoIP), web browser, and Google Maps (location provider). Energy consumption of a web browser is divided into two cases, i.e., browsing with WiFi or 3G. To compare the estimated results using the Monsoon power meter, we also show the energy consumption of the system and applications, per component energy consumption.

AppScope showed accurate estimation results in comparison to the external measurement results. As shown in Figure 10(a), the power consumption of Angry Birds showed the highest error among the five test cases. Specifically, the uniform amount of 300mW error was shown (except for the WiFi period) after the game is completely loaded, i.e. 20 seconds. CPU and LCD were continuously used in the region where high error is shown. Compared to the other four cases, we understand that the game activates N1's GPU (Integrated Graphics Processing Unit Adreno 200 on Qualcomm QSD8250 Snapdragon) and error is caused by this hardware component. To find out the exact cause of the error, we have conducted additional experiments with Android's CPU/GPU benchmark tool, StablilityTest [27]. As illustrated in Figure 10(f), while StablilityTest is preparing 3D objects to display on the screen (initial 37 seconds), the AppScope results and the results of the external power meter were nearly identical. Between 40 to 100-seconds where a 3D object was periodically rotated, there was a 300mW difference. The difference is as large as the error shown in Figure 10(a). In this region, CPU utilization was 100%. Hence, the error is assumed to be caused by the GPU operation.

In Figure 10(b) and (c), the power consumption of the WiFi interface was reflected in total energy consumption with approximately 4% error. Note that the Monsoon results in Figure 10(b) are higher than AppScope, whereas the results are opposite in Figure 10(c). We consider this to be a limitation of the linear regression-based power model that is produced by DevScope. Although the WiFi interface always operates with CPU, our power model does not consider inter-dependency of WiFi interface and CPU. This limitation may be overcome using a model that includes cross-terms, which represent the inter-dependency among components [5].

As shown in figures 10(a), (b), and (d), after 70 seconds of operation, there was a temporary increase in the energy consumption of LCD and 3G interface. These increases are generated due to an error in the data collecting program, which was implemented using SystemTap [24]. In these points, the collected workload for 2 seconds is accumulated in 1 second by a timer bug in SystemTap. In reality, there should not be a temporary increase in power consumption of LCD unless its brightness is changed. After the increase in power consumption, there was a time difference in the estimation of AppScope and Monsoon measurement.

Table 5: Energy estimation for real applications

App.	CPU (J)	WiFi (J)	GPS (J)	3G (J)	LCD (J)	System (J)	Total (J)	Monsoon (J)	Err. (%)
Angry Birds	27.4	7.1	0	0	80.3	24.0	138.8	162.7	14.7
Browser (WiFi)	28.6	14.3	0	0	82.8	25.1	150.8	144.3	4.5
Browser (3G)	25.7	0	0	36.2	85.9	13.3	161.1	174.1	7.5
Skype (WiFi)	14.8	24.6	0	0	85.0	25.7	150.1	148.8	0.9
Google Map	3.9	2.5	33.6	0	81.2	18.0	139.2	137.8	1.0

Table 5 shows each application's total and component-wise energy consumption. The total energy consumption is computed by aggregating the energy consumption of the hardware components and the system. The error is calculated using total estimated energy consumption and the results from the external power meter. All applications, with the exception of Angry Birds, showed an error rate below 7.5% during a 100-second experiment. Angry Birds showed a 14.7% error due to the aforementioned GPU operation.

7 Related Work

Recent research [7, 9, 14, 16] on smartphone power management has developed diverse power models to estimate a device's power consumption. Dong and Zhong proposed Sesame [16], which is an automatic smartphone power modeling scheme using a built-in current sensor. Their work focused on overall system power rather than power analysis on individual hardware components. This feature is hardly applicable for estimating the energy consumption of each application. Pathak et al. [14] proposed an FSM (finite state machine)-based power model using an external power measurement tool in conjunction with system call tracing. This approach may be applicable for application energy metering, but in-depth study and measurements on target devices should be required to obtain detailed power states.

Among recent works, PowerTutor [7, 8], PowerProf [9], and Eprof [29] support the estimation of application energy consumption. PowerTutor [8] is an application power estimation system that uses PowerBooter [7], which is a power model generation tool using fuel gauge sensors and knowledge of battery discharge behavior. PowerTutor [8] uses different methods to access usage statistics from *procfs* and *BatteryStat* for each hardware component. This method cannot guarantee the accuracy

of application energy consumption, due to the limitations that are discussed in Section 4.1. PowerTutor provides UID-specific energy information, but not process-specific information. Furthermore, it requires modification of the Android system software and kernel for components such as GPS and Audio. With AppScope, we use standard kernel functionalities to collect hardware usage information through an event-driven mechanism; this avoids monitoring overhead and performance degradation. In addition, AppScope provides process-specific power estimation in real-time.

Kjærgaard and Blunck proposed PowerProf [9], which is an unsupervised power profiling scheme for the smartphone using the Nokia Energy Profiler [28]. PowerProf generates component power models based on a genetic algorithm in order to automatically identify the power states of underlying hardware components. PowerProf enables online energy estimation, but the scheme is focused on power modeling rather than application energy metering. PowerProf measures power consumption for API calls issued in programming language. This method is limited in terms of application energy metering because the technique strongly depends on the programmer's intention.

Eprof [29] is a fine-grained energy profiler for smartphone applications. Based on the FSM power model [14], Eprof has the ability to analyze the asynchronous energy state of an application, modeling the tail-state energy characteristics of hardware components with routine-level granularity. Energy metering is achieved via a post-processing mechanism using an explicit accounting policy. Eprof requires modifications in the Android framework to trace the API calls; the application code, if using the Android NDK, should also be modified.

PowerScope [6] and Quanto [30] are developed towards energy estimation with hardware usage monitoring. PowerScope [6] provides detailed process-specific energy estimation for mobile devices. The scheme requires an additional computing resource, and programmers should use a set of specialized APIs to estimate power consumption. Quanto [30] is developed as a network-wide energy profiler for fast energy metering based on event-driven methods in TinyOS. The approach is similar to AppScope, which detects hardware operations in kernel, and breaks down the energy usage of a system by hardware component.

The information obtained with AppScope is closely related to energy efficient operating system research [1-5]. These works, in fact, proposed abstract OS mechanisms to limit energy that can be used by processes. The mechanism requires usage and energy consumption information regarding an application's hardware. In this context, AppScope would be useful for developing energy-aware operating systems.

8 Discussion

The accuracy of application energy metering depends on the power model of underlying hardware components. The present work used the power model of DevScope, which currently does not cope with GPU, multi-core, and memory components. Indeed, the experimental results in Figure 9(a) showed that a relatively large error is exhibited in applications using the integrated GPU. In addition to the GPU, recent smartphones are beginning to employ multi-core CPU, which necessitates the development of more advanced tools covering new hardware features. Also, the current AppScope/DevScope is limited in modeling the memory hardware component. In fact, previous work [15] showed that energy characteristics of smartphone applications differ with the nature of application; that is, CPU-bound or memory-bound jobs. We are aware that in order to model diverse hardware and obtain applications' energy consumption more accurately, both AppScope and DevScope should be supplemented with further emphasis on memory, GPU, and multi-core CPU architecture. This is, in fact, part of our future work.

Meanwhile, the tail-state energy consumption of cellular, WiFi, and GPS hardware components should be considered for fine-grained energy modeling. The Finite States Machine (FSM)-based model [14], for instance, uses power state transitions, instead of component utilization for power modeling, which enables the accurate modeling of tail-state. AppScope, however, does not detect the tail-state; hence the energy consumption on this state is not reflected in the application's energy. This limitation is fundamentally caused by the use of a linear power model in AppScope, which primarily obtains usage statistics, rather than state changes, of individual hardware components.

Although the AppScope energy metering framework includes DevScope as its core component to obtain device power models automatically and online, the core part of the AppScope framework is still the automatic acquisition of x_i^j and d_i^j for each hardware component accessed by an application. This means that the core of AppScope can practically run on any smartphone whose component power models are known *a priori* - either by DevScope or by direct measurement of individual hardware components.

9 Conclusion

In this paper, we proposed AppScope to automatically meter energy consumption of Android applications using kernel activity monitoring. AppScope traces system

calls and also analyzes Android binder IPC data. Designed as a kernel module, AppScope runs efficiently to collect fine-grained process-specific energy information. Compared to previous research on smartphone energy estimation, AppScope provides a more accurate and detailed application-specific energy estimation solution. This result will be used as an important basis in establishing a foundation to support power-related research on Android mobile devices.

Acknowledgements

We would like to thank the anonymous reviewers for their comments. A special thank you should go to our shepherd, Gernot Heiser, who has greatly helped us enhance the quality of this paper. We also appreciate the comments from Rodrigo Fonseca. This work was supported by a grant from the National Research Foundation of Korea (NRF), funded by the Korean government, Ministry of Education, Science and Technology under Grant (No.2011-0015332).

References

[1] H. Zeng, C. Ellis, A. Lebeck, and A. Vahdat. Ecosystem: Managing Energy as a First Class Operating System Resource, *ACM SIGPLAN Notices*, volume 37, pages 123–132, 2002.

[2] H. Zeng, C. Ellis, A. Lebeck, and A. Vahdat. Currentcy: A Unifying Abstraction for Expressing Energy Management Policies. In *USENIX ATC*, 2003.

[3] H. Zeng, C. Ellis, and A. Lebeck. Experiences in Managing Energy with Ecosystem. *IEEE Pervasive Computing*, 4(1):62–68, 2005.

[4] A. Roy, S. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich. Energy Management in Mobile Devices with the Cinder Operating Systemg. In *EuroSys*, 2011.

[5] D. Snowdon, E. Le Sueur, S. Petters, and G. Heiser. Koala: A Platform for OS-level Power Management. In *EuroSys*, 2009.

[6] J. Flinn and M. Satyanarayanan. Powerscope: A Tool for Profiling the Energy Usage of Mobile Applications. In *IEEE WMCSA*, 1999.

[7] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. Dick, Z. Mao, and L. Yang. Accurate Online Power Estimation and Automatic Battery Behavior based Power Model Generation for Smartphones. In *CODES+ISSS*, 2010.

[8] Powertutor, http://powertutor.org.

[9] M. B. Kjærgaard and H. Blunck. Unsupervised Power Profiling for Mobile Devices. In *Mobiquitous*, 2011

[10] Y. Xiao, R. Bhaumik, Z. Yang, M. Siekkinen, P. Savolainen, and A. Yla-Jaaski. A System-level Model for Runtime Power Estimation on Mobile Devices. In *GreenCom-CPSCom*, 2010.

[11] T. Li and L. John. Run-time Modeling and Estimation of Operating System Power Consumption. *ACM SIGMETRICS Performance Evaluation Review*, volume 31, pages 160–171, 2003.

[12] S. Gurun and C. Krintz. A Run-time, Feedback-based Energy Estimation Model for Embedded Devices. In *CODES+ISSS*, 2006.

[13] K. Singh, M. Bhadauria, and S. McKee. Real Time Power Estimation and Thread Scheduling via Performance Counters. *ACM SIGARCH Computer Architecture News*, 37(2):46–55, 2009.

[14] A. Pathak, Y. Hu, M. Zhang, P. Bahl, and Y. Wang. Fine-grained Power Modeling for Smartphones Using System Call Tracing. In *EuroSys*, 2011.

[15] A. Carroll and G. Heiser. An Analysis of Power Consumption in a Smartphone. In *USENIX ATC*, 2010.

[16] M. Dong and L. Zhong. Self-constructive High-rate System Energy Modeling for Battery-powered Mobile Systems. In *MobiSys*, 2011.

[17] Batterydiviner, https://play.google.com/store/search?q=batterydiviner.

[18] J. McCullough, Y. Agarwal, J. Chandrashekar, S. Kuppuswamy, A. Snoeren, and R. Gupta. Evaluating the Effectiveness of Model-based Power Characterization. In *USENIX ATC*, 2011.

[19] W. Jung, C. Kang, C. Yoon, D. Kim, and H. Cha. Non-intrusive and online power analysis for smartphone hardware components. *Technical Report*, MOBED-TR-2012-1, Yonsei University, 2012.

[20] Openbinder, http://www.angryredplanet.com/~hackbod/openbinder/docs/html/BinderOverview.html.

[21] Android Binder, https://www.nds.rub.de/media/attachments/files/2011/10/main.pdf.

[22] Kprobes, http://www.kernel.org/doc/Documentation/kprobes.txt.

[23] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Profiling Resource Usage for Mobile Applications: A Cross-layer Approach. In *Mobisys*, 2011.

[24] SystemTap, http://sourceware.org/systemtap.

[25] HTC Google Nexus One, http://en.wikipedia.org/wiki/Nexus_One.

[26] Monsoon, http://www.msoon.com/LabEquipment/PowerMonitor.

[27] StabilityTest, https://play.google.com/store/apps/details?id=com.into.stability.

[28] Nokia Energy Profiler, http://www.developer.nokia.com/Resources/Tools_and_downloads/Other/Nokia_Energy_Profiler/Quick_start.xhtml.

[29] A. Pathak, Y. C. Hu, and Ming Zhang. Fine Grained Energy Accounting on smartphones with Eprof. In *EuroSys*, 2012.

[30] R. Fonseca, P. Dutta, P. Levis, and I. Stoica. Quanto: Tracking Energy in Networked Embedded Systems. In *USENIX OSDI*, 2008.

Gdev: First-Class GPU Resource Management in the Operating System

Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt
Department of Computer Science, UC Santa Cruz

Abstract

Graphics processing units (GPUs) have become a very powerful platform embracing a concept of heterogeneous many-core computing. However, application domains of GPUs are currently limited to specific systems, largely due to a lack of "first-class" GPU resource management for general-purpose multi-tasking systems.

We present Gdev, a new ecosystem of GPU resource management in the operating system (OS). It allows the user space as well as the OS itself to use GPUs as first-class computing resources. Specifically, Gdev's virtual memory manager supports data swapping for excessive memory resource demands, and also provides a shared device memory functionality that allows GPU contexts to communicate with other contexts. Gdev further provides a GPU scheduling scheme to virtualize a physical GPU into multiple logical GPUs, enhancing isolation among working sets of multi-tasking systems.

Our evaluation conducted on Linux and the NVIDIA GPU shows that the basic performance of our prototype implementation is reliable even compared to proprietary software. Further detailed experiments demonstrate that Gdev achieves a 2x speedup for an encrypted file system using the GPU in the OS. Gdev can also improve the makespan of dataflow programs by up to 49% exploiting shared device memory, while an error in the utilization of virtualized GPUs can be limited within only 7%.

1 Introduction

Recent advances in many-core technology have achieved an order-of-magnitude gain in computing performance. Examples include *graphics processing units* (GPUs) – mature compute devices that best embrace a concept of heterogeneous many-core computing. In fact, TOP500 Supercomputing Sites disclosed in November 2011 [29] that three of the top five supercomputers employ clusters of GPUs as primary computing resources. Of particular note is that scientific climate applications have achieved 80x speedups leveraging GPUs [27]. Such a continuous wealth of evidence for performance benefits of GPUs has encouraged application domains of GPUs to expand to general-purpose and embedded computing. For instance, previous work have demonstrated that GPU-accelerated systems achieved an order of 10x speedups for software routers [10], 20x speedups for encrypted networks [12], and 15x speedups for motion planning [19]. This rapid growth of general-purpose computing on GPUs, *a.k.a.*, GPGPU, is thanks to emergence of new programming languages, such as CUDA [21].

Seen from these trends, GPUs are becoming more and more applicable for general-purpose systems. However, system software support for GPUs in today's market is tailored to accelerate particular applications dedicated to the system; it is not well-designed to integrate GPUs into general-purpose *multi-tasking* systems. Albeit speedups of individual application programs, the previous research raised above [10, 12, 19] could not provide performance or quality-of-service (QoS) management without system software support. Given that networked and embedded systems are by nature composed of multiple clients and components, it is essential that GPUs should be managed as first-class computing resources so that various tasks can access GPUs concurrently in a reliable manner.

The research community has articulated the needs of enhancement in the operating system (OS) [2, 15, 24], hypervisor [9], and runtime library [14] to make GPUs available in interactive and/or virtualized multi-tasking environments. However, all these pieces of work depend highly on the user-space runtime system, often included as part of proprietary software, which provides the user space with an application programming interface (API). This framework indeed limits the potential of GPUs to the user space. For example, it prevents the file system or network stack in the OS from using GPUs directly. There is another issue of concern with this framework: the device driver needs to expose resource management primitives to the user space, since the runtime system is employed in the user space, implying that non-privileged user-space programs may abuse GPU resources. As a matter of fact, we can launch any program on an NVIDIA GPU without using any user-space runtime libraries, but using an `ioctl` system call directly. This explains that GPUs should be protected by the OS as well as CPUs.

In addition to those conceptual issues, there exist more fundamental and practical issues with publicly-available GPGPU software. For example, memory allocation for GPU computing is not allowed to exceed the physical capacity of device memory. We are also not aware of any API that allows GPU contexts to share memory resources

with other contexts. Such programming constraints may not be acceptable in general-purpose systems.

Contribution: We present **Gdev**, a new approach to GPU resource management in the OS that addresses the current limitations of GPU computing. Gdev integrates runtime support for GPUs into the OS, which allows the user space as well as the OS itself to use GPUs with the identical API set, while protecting GPUs from non-privileged user-space programs at the OS level. Building on this runtime-unified OS model, Gdev further provides first-class GPU resource management schemes for multi-tasking systems. Specifically, Gdev allows programmers to share device memory resources among GPU contexts using an explicit API. We also use this shared memory functionality to enable GPU contexts to allocate memory exceeding the physical size of device memory. Finally, Gdev is able to virtualize the GPU into multiple logical GPUs to enhance isolation among working sets of multi-tasking systems. As a proof of concept, we also provide an open-source implementation of Gdev. To summarize, this paper makes the following contributions:

- Identifies the advantage/disadvantage of integrating runtime support for GPUs into the OS.
- Enables the OS itself to use GPUs.
- Makes GPUs "first-class" computing resources in multi-tasking systems – memory management for inter-process communication (IPC) and scheduling for GPU virtualization.
- Provides open-source implementations of the GPU device driver, runtime/API libraries, utility tools, and Gdev resource management primitives.
- Demonstrates the capabilities of Gdev using real-world benchmarks and applications.

Organization: The rest of this paper is organized as follows. Section 2 provides the model and assumptions behind this paper. Section 3 outlines the concept of Gdev. Section 4 and 5 present Gdev memory management and scheduling schemes. Section 6 describes our prototype implementation, and Section 7 demonstrates our detailed experimental results. Section 8 discusses related work. We provide our concluding remarks in Section 9.

2 System Model

This paper focuses on a system composed of a GPU and a multi-core CPU. GPU applications use a set of the API supported by the system, typically taking the following steps: (i) allocate space to device memory, (ii) copy data to the allocated device memory space, (iii) launch the program on the GPU, (iv) copy resultant data back to host memory, and (v) free the allocated device memory space. We also assume that the GPU is designed based on

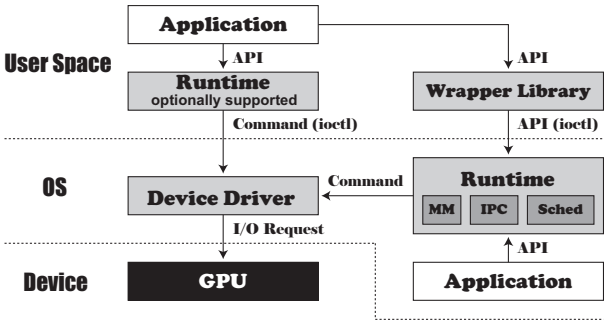


Figure 1: Logical view of Gdev's ecosystem.

NVIDIA's *Fermi* architecture [20]. The concept of Gdev, however, is not limited to Fermi, but is also applicable to those based on the following model.

Command: The GPU operates using the architecture-specific commands. Each GPU context is assigned with a FIFO queue to which the program running on the CPU submits the commands. Computations and data transfers on the GPU are triggered only when the corresponding commands are dispatched by the GPU itself.

Channel: Each GPU context is assigned with a GPU hardware channel within which command dispatching is managed. Fermi does not permit multiple channels to access the same GPU functional unit simultaneously, but allow them to coexist being switched automatically in hardware. This constraint may however be removed in the future architectures or product lines.

Address Space: Each GPU context is assigned with virtual address space managed through the page table configured by the device driver. Address translations are performed by he memory management unit on the GPU.

Compute Unit: The GPU maps *threads* assigned by programmers to *cores* on the compute unit. This thread assignment is not visible to the system, implying that GPU resource management at the system level should be context-based. Multiple contexts cannot execute on the compute unit at once due to the channel constraint, but multiple requests issued from the same context can be processed simultaneously. We also assume that GPU computation is non-preemptive.

DMA Unit: There are two types of DMA units for data transmission: (i) synchronous with the compute unit and (ii) asynchronous. Only the latter type of DMA units can overlap their operations with the compute unit. We also assume that DMA transaction is non-preemptive.

3 Gdev Ecosystem

Gdev aims to (i) enhance GPU resource management and (ii) extend a class of applications that can leverage GPUs. To this end, we integrate the major portion of runtime support into the OS. Figure 1 illustrates the logical view

of Gdev's ecosystem. For a compatibility issue, we still support the conventional stack where applications make API calls to the user-space runtime library, but system designers may disable this stack to remove the concern discussed in Section 1. A new ecosystem introduced by Gdev is runtime support integrated in the OS, allowing the user space as well as the OS to use the identical API set. This ecosystem prevents non-privileged user-space programs from bypassing the runtime system to access GPUs. The wrapper library is a small piece of software provided for user-space applications, which relays API calls to the runtime system employed in the OS.

Leveraging this ecosystem, we design an API-driven GPU resource management scheme. Figure 1 shows that Gdev allows the OS to manage API calls, whereas the traditional model translates API calls to GPU commands before the OS receives them. As discussed in previous work [15], it is very hard to analyze GPU commands and recognize the corresponding API calls in the OS. Hence, the existing GPU resource management schemes in the OS [2, 15] compromise overhead to invoke the scheduler at every GPU command submission, unless an additional programming abstraction is provided [24]. On the other hand, Gdev can manage GPU resources along with API calls, without any additional programming abstractions.

Programming Model: We provide a set of low-level functions for GPGPU programming, called "Gdev API". Gdev API is a useful backend for high-level APIs, such as CUDA. The details of Gdev API can be found at our project website [25]. Programmers may use either Gdev API directly or high-level APIs built on top of Gdev API. This paper particularly assumes that programmers use the well-known CUDA Driver API 4.0 [21].

Gdev uses an existing programming framework and commodity compiler, such as NVIDIA CUDA Compiler (NVCC) [21]. When a program is compiled, two pieces of binary are generated. One executes on the CPU, and loads the other binary onto the GPU. The CPU binary is provided as an executable file or loadable module, while the GPU binary is an object file. Hence, both user-space and OS-space applications can use the same framework: (i) read the GPU binary file and (ii) load it onto the GPU. The detailed information embedded in the object file, such as code, static data, stack size, local memory size, and parameter format, may depend on the programming language, but the framework does not depend on it once the object file is parsed.

Resource Management: We provide device memory management and GPU scheduling schemes to manage GPUs as first-class computing resources. Especially we realize shared device memory for IPC, data swapping for large memory demands, resource-based queuing for throughput, and bandwidth-aware resource partitioning for isolation of virtual GPUs. Since some pieces of these features require low-level access to system information, such as I/O space, DMA pages, and task control blocks, it is not straightforward for traditional user-space runtime systems to realize such a resource management scheme. Therefore, we claim that Gdev is a suitable approach to first-class GPU resource management. The concept of Gdev is also not limited to GPUs, but can be generalized for a broad class of heterogeneous compute devices.

4 Device Memory Management

Gdev manages device memory using the virtual memory management unit supported by the GPU. Virtual address space for GPU contexts can be set through the page table. Gdev stores this page table in device memory, though it can also be stored in host memory. Beyond such basic pieces of memory management, this section seeks how to improve memory-copy throughput. We also explore how to share memory resources among GPU contexts, and support data swap for excessive memory demands.

4.1 Memory-Copy Optimization

Given that data move across device and host memory back and forth, memory-copy throughput could govern the overall performance of GPU applications. While the primary goal of this paper is to enhance GPU resource management, we respect standalone performance as well for practical use. Hence, we first study the characteristic of memory-copy operations.

Split Transaction: We often need to copy the same data set *twice* to communicate with the GPU, unless we allocate buffers to host I/O memory directly. One copy happens within host memory, moving data between main memory and host I/O memory, *a.k.a.*, pinned pages of host memory. The other copy happens between device and host I/O memory. In order to optimize this two-stage memory-copy operation, we split the data buffer into a fixed size of multiple chunks. Using split transactions, while some chunk is transferred within host memory, the preceding chunk can be transferred between device and host I/O memory. Thus, only the first and last pieces of chunks need to be transferred alone, and other chunks are all overlapped, thus reducing a total makespan almost half. An additional advantage of this method is that only the same size of an intermediate "bounce" buffer as the chunk size is required on host I/O memory, thus reducing the usage of host I/O memory significantly. It should be noted that "pinned" pages do not use split transaction.

Direct I/O Access: The split transaction is effective for a large size of data. For a small size of data, however, the use of DMA engines incurs non-trivial overhead by itself. Hence, we also employ a method to read/write data one by one by mapping device memory space onto host I/O memory space, rather than send/receive data in

burst mode by using DMA engines. We have found that direct I/O access is much faster than DMA transaction for a small size of data. In Section 7, we will identify a boundary on the data size that inverts the latency of I/O access and DMA, and also derive the best chunk size to optimize memory-copy throughput.

4.2 Shared Device Memory

Existing GPU programming languages do not support an explicit API for IPC. For example, data communications among GPU contexts incur significant overhead due to copying data back and forth between device and host memory. Currently, an OS dataflow abstraction [24] is a useful approach to reduce such data movement costs; users are required to use a dataflow programming model. We believe that it is more flexible and straightforward for programmers to use a familiar POSIX-like method.

Gdev supports a set of API functions to share device memory space among GPU contexts respecting POSIX IPC functions of `shmget`, `shmat`, `shmdt`, and `shmctl`. As a high-level API, we extend CUDA to provide new API functions of `cuShmGet`, `cuShmAt`, `cuShmDt`, and `cuShmCtl` in our CUDA implementation so that CUDA applications can easily leverage Gdev's shared device memory functionality.

Our shared memory design is straightforward, though its implementation is challenging. Suppose that we use the above extended CUDA API for IPC. Upon the first call to `cuShmGet`, Gdev allocates new space to device memory, and holds an identifier to this memory object. After the first call, Gdev simply returns this identifier to this call. When `cuShmAt` is called, the allocated space is mapped to the virtual address space of the corresponding GPU context. This address mapping is done by setting the page table so that the virtual address points to the physical memory space of this shared memory object. The allocated space can be unmapped by `cuShmDt` and freed by `cuShmCtl`. If the shared memory object needs exclusive access, the host program running on the CPU must use traditional mutex and semaphore primitives.

4.3 Data Swapping

We have found that proprietary software in Linux [21] fails to allocate device memory exceeding the physical memory capacity, while the Windows display driver [23] supports data swapping to some extent. In either case, however, a framework of data swapping with GPUs has not been well studied so far. This section explores how to swap data in the presence of multiple GPU contexts.

Gdev uses the shared device memory functionality to achieve data swapping. When memory allocation fails due to a short of free memory space, Gdev seeks memory objects whose allocated size is greater than the requested size, and selects one owned by a low-priority context, where ties are broken arbitrarily. This "victim" memory object is shared by the caller context *implicitly*. Unlike an explicit shared memory object obtained through the API presented in Section 4.2, an implicit shared memory object must evict data when accessed by other contexts, and retrieve them later when the corresponding context is resumed. Since Gdev is designed API-driven, it is known when contexts may access the shared memory object:

- The memory-copy API will affect specific address space given by the API parameters. Hence, we need to evict only such data that cover this range.
- The compute-launch API may also be relevant to some address space, but its address range is not all specified when the API is called, since the program may use dynamic memory allocation. Hence, we need to evict such data that are associated with all the memory objects owned by the context.

We allocate swap buffers to host main memory for evicted data. Swapping itself is a simple asynchronous memory-copy operation, but is not visible to application programs. It should be noted that swapping never occurs when copying data from device to host memory. If the corresponding data set is evicted in the swap space, it can be retrieved from the swap space directly, and there is no need to swap it back to device memory.

Reducing Latency: It is apparent that the swapping latency could be non-trivial, depending on the data size. In order to reduce this latency, Gdev reserves a certain amount of device memory space as *temporal swap space*. Since a memory-copy operation within device memory is much faster than that between device and host memory, Gdev first tries to evict data to this temporal swap space. This temporarily-evicted data set is eventually evicted to host memory after a while to free up the swap space for other contexts. Gdev also tries to hide this second eviction latency by overlapping it with GPU computation launched by the same context. We create a special GPU context that is dedicated to memory-copy operations for eviction, since the compute and DMA units cannot be used by the same context simultaneously. This approach is quite reasonable because data eviction is likely to be followed by GPU computation. Evicted data, if exist, must be retrieved before GPU computation is launched. If they remain in the swap space, they can be retrieved at low cost. Else, Gdev retrieves them from host memory.

5 GPU Scheduling

The goal of the Gdev scheduler is to correctly assign computation and data transmission times for each GPU context based on the given scheduling policy. Although we make use of some previous techniques [14, 15], Gdev

provides a new queuing scheme and virtual GPU support for multi-tasking systems. Gdev also propagates the task priority used in the OS to the GPU context.

5.1 Scheduling and Queuing

Gdev uses a similar scheme to TimeGraph [15] for GPU scheduling. Specifically, it allows GPU contexts to use GPU resources only when no other contexts are using the corresponding resources. The stalling GPU contexts are queued by the Gdev scheduler while waiting for the current context to leave the resources. In order to notify the completion of the current context execution, Gdev uses additional GPU commands to generate an interrupt from the GPU. Upon every interrupt, the highest-priority context is dispatched to the GPU from the waiting queue. Computation and data transmission times are separately accumulated for resource accounting. For computations, we allow the same context to launch multiple compute instances simultaneously, and the total makespan from the first to the last instance is deemed as the computation time. PTask [24] and RGEM [14] also provide similar schedulers, but do not use interrupts, and hence resource accounting is managed by the user space via the API.

Gdev is API-driven where the scheduler is invoked only when computation or data transmission requests are submitted, whereas TimeGraph is command-driven, which invokes the scheduler whenever GPU commands are flushed. In this regard, Gdev is similar to PTask [24] and RGEM [14]. However, Gdev differs from these two approaches in that it can separate queues for accounting of computations and data transmissions, which we call the *Multiple Resource Queues* (MRQ) scheme. On the other hand, what we call the *Single Device Queue* (SDQ) scheme uses a single queue per device for accounting.

The MRQ scheme is apparently more efficient than the SDQ scheme, when computations and data transmissions can be overlapped. Suppose that there are two contexts both requesting 50% of computation and 50% of data transmission demands. The SDQ scheme presumes that the demand of each context is 50 + 50 = 100%, implying a total demand of 200% by the two contexts. As a result, this workload looks overloaded under the SDQ scheme. The MRQ scheme, on the other hand, does not consider the total workload to be overloaded due to overlapping but each resource to be fully utilized.

Gdev creates two different scheduler threads to control the resource usage of the GPU compute unit and DMA unit separately. The compute scheduler thread is invoked by GPU interrupts generated upon the completion of each GPU compute operation, while the DMA scheduler thread is awakened by the Gdev runtime system when the memory-copy operation is completed, since we do not use interrupts for memory-copy operations.

```
vgpu->bgt: budget of the virtual GPU.
vgpu->utl: actual GPU utilization of the virtual GPU.
vgpu->bw: bandwidth assigned to the virtual GPU.
current/next: current/next virtual GPU selected for run.

void on_arrival(vgpu, ctx) {
  if (current && current != vgpu)
    suspend(ctx);
  dispatch(ctx);
}
VirtualGPU on_completion(vgpu, ctx) {
  if (vgpu->bgt < 0 && vgpu->utl > vgpu->bw)
    move_to_queue_tail(vgpu);
  next = get_queue_head();
  if (!next) return null;
  if (next != vgpu && next->utl > next->bw) {
    wait_for_short();
    if (current) return null;
  }
  return next;
}
```

Figure 2: Pseudo-code of the BAND scheduler.

5.2 GPU Virtualization

Gdev is able to virtualize a physical GPU into multiple logical GPUs to protect working groups of multi-tasking systems from interference. Virtual GPUs are activated by specifying weights of GPU resources assigned to each of them. GPU resources are classified to *memory share*, *memory bandwidth*, and *compute bandwidth*. Memory share is the weight of physical memory available for the virtual GPU. Memory bandwidth is the amount of time in a certain period allocated for memory-copy operations using the virtual GPU, while compute bandwidth is that for compute operations. Regarding memory share, Gdev simply partitions physical memory space. Meanwhile, we provide the GPU scheduler to meet the requirements of compute and memory-copy bandwidth. Considering similar characteristics of non-preemptive computations and data transmissions, we apply the same policy to the compute and memory-copy schedulers.

The challenge for virtual GPU scheduling is raised by the non-preemptive and burst nature of GPU workloads. We have implemented the Credit scheduling algorithm supported by Xen hypervisor [1] to verify if an existing virtual CPU scheduling policy can be applied for a virtual GPU scheduler. However, we have found that the Credit scheduler fails to maintain the desired bandwidth for the virtual GPU, largely attributed to the fact that it presumes preemptive constantly-working CPU workloads, while GPU workloads are non-preemptive and bursting.

To overcome the virtual GPU scheduling problem, we propose a *bandwidth-aware non-preemptive device* (BAND) scheduling algorithm. The pseudo-code of the

BAND scheduler is shown in Figure 2. The `on_arrival` function is called when a GPU context (`ctx`) running on a virtual GPU (`vgpu`) attempts to access GPU resources for computations or data transmissions. The context can be dispatched to the GPU only if no other virtual GPUs are using the GPU. Otherwise, the corresponding task is suspended. The `on_completion` function is called by the scheduler thread upon the completion of the context (`ctx`) assigned to the virtual GPU (`vgpu`), selecting the next virtual GPU to operate.

The BAND scheduler is based on the Credit scheduler, but differs in the following two points. First, the BAND scheduler lowers the priority of the virtual GPU, when its budget (credit) is exhausted *and* its actual utilization of the GPU is exceeding the assigned bandwidth, whereas the Credit scheduler always lowers the priority, when the budget is exhausted. This prioritization compensates for credit errors posed due to non-preemptive executions.

The second modification to the Credit scheduler is that the BAND scheduler waits for a certain amount of time specified by the system designer, if the GPU utilization of the virtual GPU selected by the scheduler is exceeding its assigned bandwidth. This "time-buffering" approach works for non-preemptive burst workloads. Suppose that the system has two virtual GPUs, both of which run some burst-workload GPU contexts, but their non-preemptive execution times are different. If the contexts arrive in turn, they are also dispatched to the GPU in turn, but the GPU utilization could not be fair due to different lengths of non-preemptive executions. If the scheduler waits for a short interval, however, the context with a short length of non-preemptive execution could arrive with the next request, and the `on_arrival` function can dispatch it to the GPU while the scheduler is waiting. Thus, resource allocations could become fairer. In this case, we need not to select the next virtual GPU, since the `on_arrival` function has already dispatched one. If no contexts have arrived, however, we return the selected virtual GPU. This situation implies that there are no burst workloads, and hence no emergency to meet the bandwidth.

6 System Implementation

Our prototype implementation is fully open-source and available for NVIDIA Fermi GPUs with the Linux kernel 2.6.33 or later, without any kernel modifications. It does not depend on proprietary software except for compilers. Hence, it is well self-contained and easy-to-use.

Interface: Gdev is a Linux kernel module (a character device driver) composing the device driver and runtime library. The device driver manages low-level hardware resources, such as channels and page tables, to operate the GPU. The runtime library manages GPU commands and API calls. It directly uses the device-driver functions to control hardware resource usage for first-class GPU resource management. The Gdev API is implemented in this runtime library. The kernel symbols of the API functions are exported so that other OS modules can call them. These API functions are also one-to-one mapped to the `ioctl` commands defined by Gdev so that user-space programs can also be managed by Gdev.

We provide two versions of CUDA Driver API: one for the user space and the other for the OS. The former is provided as a typical user-space library, while the latter is provided as a kernel module, called *kcuda*, which implements and exports the CUDA API functions. They however internally use Gdev API to access the GPU.

We use `/proc` filesystem in Linux to configure Gdev. For example, the number of virtual GPUs and their maps to physical GPUs are visible to users through `/proc`. The compute and memory bandwidth and memory share for each virtual GPU are also configurable at runtime through `/proc`. We further plan to integrate the configuration of priority and reserve for each single task into `/proc`, using the TimeGraph approach [15].

Gdev creates the same number of character device files as virtual GPUs, *i.e.*, /dev/{gdev0,gdev1,...}. When users open one of these device files using Gdev API or CUDA API, it behaves as if it were one for the physical GPU.

Resource Parameters: The performance of Gdev is governed by resource parameters, such as the page size for virtual memory, temporal swap size, waiting time for the Band scheduler, period for virtual GPU budgets, chunk size for memory-copy, and boundary between I/O access and DMA. We use a page size of 4KB, as the Linux kernel uses the same page size for host virtual memory by default. The swap size is statically set 10% of the physical device memory. The waiting time for the Band scheduler is also statically set 500 microseconds. For the period of virtual GPU budgets, we respect Xen's default setup, *i.e.*, we set it 30ms. The rest of resource parameters will be determined in Section 7.

Portability: We use Direct Rendering Infrastructure (DRI) [18] – a Linux framework for graphics rendering with the GPU – to communicate with the Linux kernel. Hence, some Gdev functionality may be used to manage not only compute but also 3-D graphics applications. Our implementation approach also abstracts GPU resources by device, address space, context, and memory objects, which allows other device drivers and GPU architectures to be easily ported.

Limitations: Our prototype implementation is still partly experimental. In particular, it does not yet support texture and 3-D processing. Hence, our CUDA Driver API implementation is limited to some extent, but many CUDA programs can execute with this limited set of functions, as we will demonstrate in Section 7. CUDA Runtime API [21], a more high-level API than CUDA

Driver API, is also not supported yet, but we could use Ocelot [5] to translate CUDA Runtime API to CUDA Driver API. Despite such limitations, we believe that our prototype implementation contributes greatly to future research on GPU resource management, given that open-source drivers/runtimes for GPUs are very limited today.

7 Experimental Evaluation

We evaluate our Gdev prototype, using the Rodinia benchmarks [3], GPU-accelerated eCryptfs encrypted file system from KGPU [28], FAST database search [16], and some dataflow microbenchmarks from PTask [24]. We disclose that the basic performance of our prototype is practical even compared to proprietary software, and also demonstrate that Gdev provides significant benefits for GPU applications in time-sharing systems.

Our experiments are conducted with the Linux kernel 2.6.39 on NVIDIA GeForce GTX 480 graphics card and Intel Core 2 Extreme QX9650 processor. GPU programs are written in CUDA and compiled by NVCC [21], while CPU programs are compiled by gcc 4.4.6.

7.1 Basic Performance

We evaluate the standalone performance of applications achieved by our Gdev prototype to argue that the rest of our evaluation is practical in the real world. To this end, first of all, we need to find the best parameters used for memory-copy optimization, using simple test code that copies data between device and host memory.

Figure 3 shows the impact of the chunk size on data transfer times for host-to-device (HtoD) and device-to-host (DtoH) directions respectively, when using DMA-based memory-copy operations with 256MB and 512MB of data. Since each chunk incurs some overhead in DMA configuration, a smaller chunk size producing a greater number of chunks increases a transfer time. On the other hand, there is a constraint that the first and last pieces of chunks cannot be overlapped with others, as described in Section 4.1. Hence, a larger chunk size leading to a longer blocking time with these pieces of chunks also increases a transfer time. According to our observation, a chunk size of 4MB is the best trade-off for both HtoD and DtoH directions. We therefore set the chunk size to 4MB for our experiments.

Figure 4 shows the relative speed of direct I/O access to DMA for a small size of data. Due to some hardware effect, HtoD and DtoH directions show different transfer times, but it clearly explains the advantage of direct I/O access for small data. According to our observation, the data transfer speed inverses around a data size of 4KB and 1KB for HtoD and DtoH directions respectively. We therefore set the boundary of direct I/O access and DMA to 4KB and 1KB for them respectively.

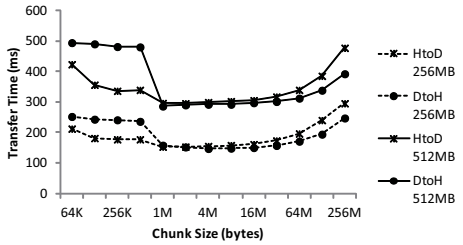


Figure 3: Impact of the chunk size on DMA speeds.

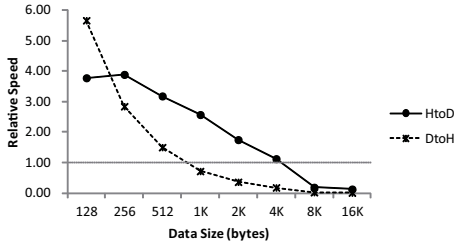


Figure 4: Relative speed of I/O access to DMA.

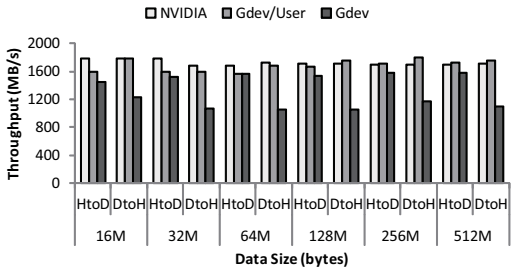


Figure 5: Memory-copy throughput.

Figure 5 shows memory-copy throughput achieved by our Gdev prototype compared to NVIDIA's proprietary software. "Gdev/User" employs a runtime library in the user-space, while "Gdev" integrates runtime support into the OS. Interestingly, user-space runtime achieves higher throughput than OS-space runtime, particularly for DtoH direction. This difference comes from `memcpy`'s effect within host memory. In fact, the `memcpy` implementation in the Linux kernel is slower than that in the user-space GNU library, when copying data from host I/O to main memory. This could be a disadvantage of our approach. We will investigate this effect more in depth. Apart from the DtoH memory-copy throughput, however, our Gdev prototype and NVIDIA's proprietary software are almost competitive.

Figure 6 demonstrates the standalone performance of benchmarks achieved by our Gdev prototype compared to NVIDIA's proprietary software. Table 1 describes the microbenchmarks and Rodinia [3] benchmarks used in this evaluation. First of all, we have found that NVIDIA GPUs have some "performance mode" to boost hardware performance that we do not use for our Gdev prototype implementation. As observed in the LOOP benchmark result, our Gdev prototype incurs about 20% of decrease

Table 1: List of benchmarks.

Benchmark	Description
LOOP	Long-loop compute without data
MADD	1024x1024 matrix addition
MMUL	1024x1024 matrix multiplication
CPY	256MB of HtoD and DtoH
PINCPY	CPY using pinned host I/O memory
BP	Back propagation (pattern recognition)
BFS	Breadth-first search (graph algorithm)
HW	Heart wall (medical imaging)
HS	Hotspot (physics simulation)
LUD	LU decomposition (linear algebra)
NN	K-nearest neighbors (data mining)
NW	Needleman-wunsch (bioinformatics)
SRAD	Speckle reducing anisotropic diffusion (imaging)
SRAD2	SRAD with random pseudo-inputs (imaging)

in performance compared to the proprietary software due to a lack of performance mode. However, the impact of performance mode is workload dependent. If workloads are very compute-intensive, such as the HW and SRAD benchmarks, this impact appears clearly, whereas some friendly workloads, such as the BFS and HS benchmarks, are not influenced very much. In either case, however, this is an implementation issue, but is not a conceptual limitation of Gdev. These benchmark results also imply that Gdev's runtime-unified OS approach would not be appreciated by data-intensive workloads. For example, the BP benchmark is associated with a very large size of data, though its compute demand is not very high. This type of workload would not perform well with our Gdev prototype, since the `memcpy` function of the Linux kernel becomes a bottleneck. On the other hand, the PINCPY benchmark does not need `memcpy` operations. Hence, performance does not depend on implementations.

7.2 Reliability

We next evaluate reliability of runtime support integrated in the OS. Figure 7 compares the performances of the OS-space API-driven scheme (Gdev and PTask [24]), the OS-space command-driven scheme (TimeGraph [15] and GERM [2]), and the user-space API-driven scheme (RGEM [14]). We run Rodinia benchmarks recursively as fast as possible as real-time tasks, contending with such background tasks that bypass the user-space library and launch many meaningless GPU commands. The user-space API-driven scheme severely suffers from this situation, since it cannot schedule these bypassing tasks at all. The OS-space command-driven scheme is able to sustain the interference to some extent by using the GPU command scheduler, but the overhead is non-trivial due to many scheduler invocations. On the other hand, the OS-space API-driven scheme can reject such command submission that is not submitted through the API. Gdev and PTask are both API-driven, but PTask exposes the

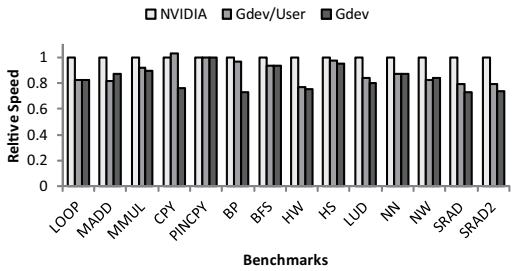


Figure 6: Basic standalone performance.

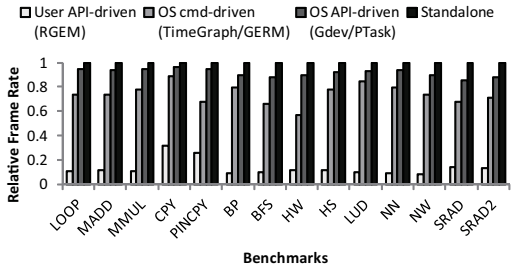


Figure 7: Unconstrained real-time performance.

system call to user-space programs, which could allow misbehaving tasks to abuse GPU resources. This evinces that Gdev's approach that integrates runtime support into the OS is a more reliable solution.

7.3 GPU Acceleration for the OS

We now evaluate the performance of the Linux encrypted file system accelerated by the GPU. In particular, we use KGPU's implementation of eCryptfs [28]. KGPU is a framework that allows the OS to access the user-space runtime library to use GPUs for computations. We have modified KGPU's eCryptfs implementation to call the CUDA API functions provided by Gdev directly instead of sending requests to the KGPU user-space daemon.

Figure 8 and 9 show the read and write throughput of several versions of eCryptfs. "CPU" represents the CPU implementation, while "KGPU & NVIDIA" and "KGPU & Gdev/User" represent those using KGPU with NVIDIA's library and Gdev's library respectively in the user space. "Gdev" is our contribution that enables the eCryptfs module to use the GPU directly within the OS. Due to some page cache effect, read and write are not identical in throughput, but an advantage of using the GPU is clearly depicted. One may observe that Gdev's runtime-unified OS approach does not really outperform KGPU's approach. This is not surprising at all, because a magnitude of improvements in latency achieved by our OS approach would be at most microseconds, while the AES/DES operations of eCryptfs performed on the GPU are orders-of-milliseconds. Nonetheless, Gdev provides a significant benefit that the OS is freed from the user space, and thus is more secure.

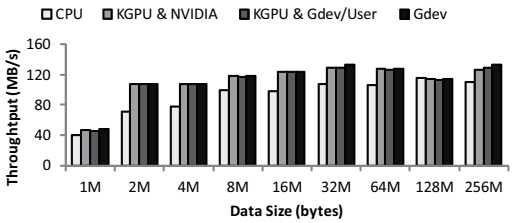


Figure 8: eCryptfs read throughput.

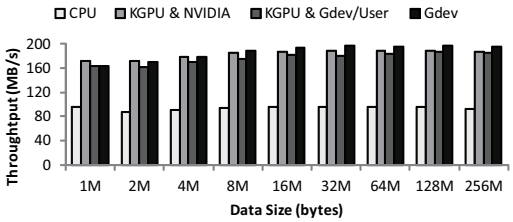


Figure 9: eCryptfs write throughput.

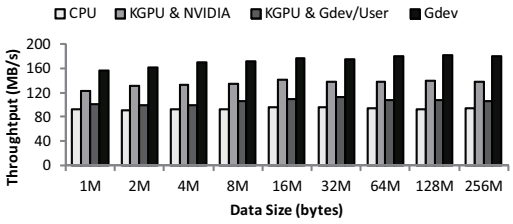


Figure 10: eCryptfs write throughput with priorities.

A further advantage of using Gdev appears in a multi-tasking scenario. Figure 10 shows the write throughput of eCryptfs when the FAST search task [16] is competing for the GPU. Since Gdev supports priorities in the OS, we can assign the eCryptfs task with the highest priority, while the search task is still assigned a higher priority than other tasks. Using KGPU in this scenario, however, the performance of eCryptfs is affected by the search task due to a lack of prioritization, as observed in "KGPU & NVIDIA". Even with priorities, KGPU could suffer from a priority inversion problem, where the high-priority eCryptfs task is reduced to the KGPU priority level when accessing the GPU, while the search task is executing at the higher priority level. We could assign a high priority to the user-space KGPU daemon to avoid this priority inversion problem, but it affects all user-space GPU applications performance. On the other hand, Gdev can assign each GPU application with an identical priority, which addresses the priority inversion problem fundamentally.

7.4 Effect of Shared Device Memory

Figure 11 shows the speedups of dataflow benchmarks brought by Gdev's shared device memory functionality. Respecting PTask's setup [24] for a similar evaluation, we make a dataflow by a 6x32 tree or a 6x10 rectangle.

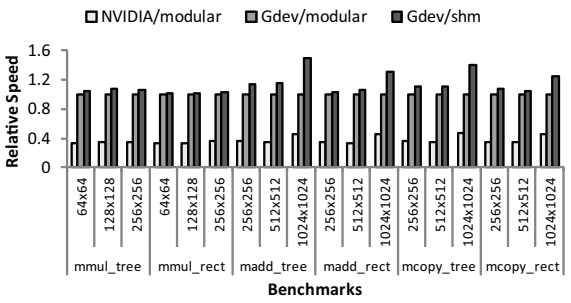


Figure 11: Impact of shared memory on dataflow tasks.

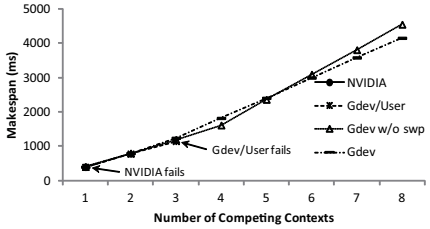


Figure 12: Impact of swapping latency.

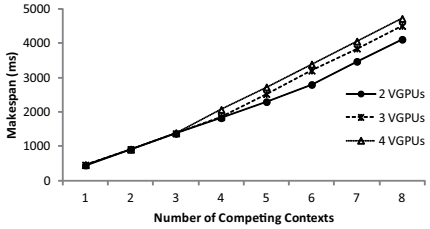
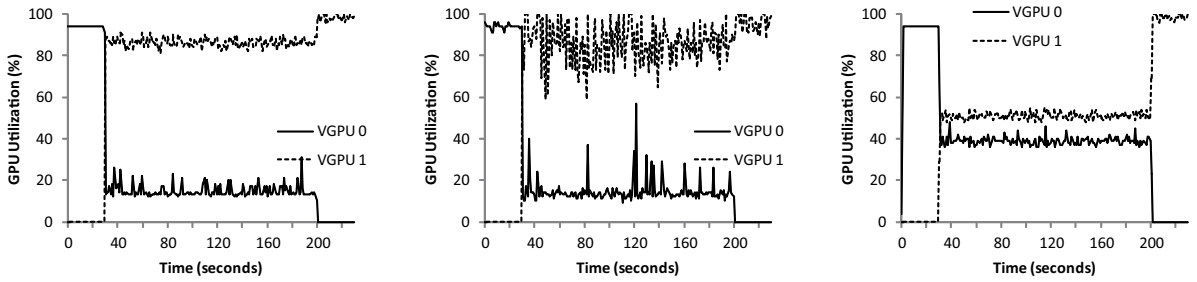


Figure 13: Impact of swapping latency on virtual GPUs.

"NVIDIA/modular" and "Gdev/modular" use NVIDIA's and Gdev's CUDA implementations respectively, where a dataflow program is implemented in such a way that allocates a self-contained context to each graph node as a module, and connects its output and input by copying data between host and device memory back and forth. "Gdev/shm" uses shared device memory, *i.e.*, it connects output and input by sharing the same "key" associated with the same memory space. According to the results, shared device memory is fairly effective for dataflows with large data. For example, it gains a 49% speedup for the 1024x1024 madd tree. Specifically, "Gdev/modular" took 1424ms while "Gdev/shm" took 953ms to complete this dataflow. This indeed makes sense. The average data transfer time for a 1024x1024 integer value was about 8ms, and we can reduce data communications by a total of 32+16+8+4+2=62 intermediate nodes for a 6x32 tree, which results in a total reduced time of 8x62=496ms. It should be noted that PTask achieves more speedups due to advanced dataflow scheduling [24]. However, we provide users with a first-class API primitive to manage shared device memory, which could be used as a generic IPC method to address different problems. Therefore, we distinguish our contribution from PTask. In addition, it



(a) FIFO scheduler (b) Credit scheduler (c) BAND scheduler

Figure 14: Util. of virtual GPUs under unfair workloads.

is surprising that our prototype system outperforms the proprietary software significantly. We suspect that the proprietary one takes a long time to initialize contexts when there are many active contexts, though an in-depth investigation is required.

Figure 12 depicts the impact of memory swapping on the makespan of multiple 128MB-data FAST search tasks, when another 1GB-data FAST search task runs at the highest priority level. Given that the GPU used in this evaluation supports 1.6GB of device memory, we cannot create more than three 128MB-data search tasks at once if memory swapping is not provided. Memory swapping uses shared device memory, which needs access to the page table. Hence, our prototype implementation does not support memory swapping as well as shared device memory for "Gdev/User", and it fails when the number of the small search tasks exceeds three. It is interesting to observe that NVIDIA' proprietary software fails when the number of the small search tasks exceeds one. This is because NVIDIA's proprietary software reserves some amount of device memory for other purposes. Unlike the user-space runtime approaches, Gdev's runtime-unified OS approach can support memory swapping, and all the 128MB-data search tasks can survive under this memory pressure. However, a reflection point where the slope of increase in the makespan changes is different, depending on whether the temporal swap space allocated on device memory is used or not. When the temporal swap space is not used, the reflection point is clearer as observed in "Gdev w/o swp", because the swapping latency is not trivial due to data movement between host and device memory. Using the temporal swap space, on the other hand, we can reduce the impact of memory swapping on the makespan of the search tasks, but the reflection point appears slightly earlier, since the temporal swap space itself occupies certain space on device memory.

Figure 13 shows the impact of memory swapping on virtual GPUs. In this experiment, we introduce virtual GPUs, and execute 128MB-data search tasks on the first virtual GPU. The memory size available for the virtual GPU is more restricted in the presence of more virtual GPUs. We confirm that the makespans become longer and their reflection points appear earlier for a greater number of virtual GPUs, but all the search tasks can still complete. This explains that memory swapping is also useful on virtual GPUs.

7.5 Isolation among Virtual GPUs

We now evaluate Gdev in terms of the isolation among virtual GPUs. Figure 14 demonstrates the actual GPU utilization of two virtual GPUs, achieved by the FIFO, Credit, and BAND schedulers under the SDQ scheme. VGPU 0 executes the LUD benchmark to produce short-length tasks, while VGPU 1 executes the HW benchmark to produce long-length tasks. These tasks run repeatedly for 200 seconds to impose high workloads on the entire system. To see a workload change clearly, VGPU 1 is started 30 seconds after VGPU 0. Our observation is that the FIFO scheduler is not capable of enforcing isolation at all. The Credit scheduler also fails to provide isolation, since it is not designed to handle non-preemptive burst workload. The BAND scheduler, however, can almost provide the desired GPU utilization, thanks to the time-buffering policy that allows short-length tasks to meet the assigned bandwidth. An error in the utilization of two virtual GPUs is retained within 7% on average.

We next study the effectiveness of the MRQ scheme that separates the queues for compute and memory-copy operations. Figure 15 illustrates the utilization of two virtual GPUs under the BAND scheduler, executing the SRAD benchmark tasks with different sizes of image. We noticed that the compute and memory-copy operations can be overlapped, but they affect the run-to-completion time with each other. When VGPU 1 uses more compute resources due to a large size of computation, the length of memory-copy operations requested by VGPU 0 is prolonged due to overlapping. As a result, it requires more memory-copy bandwidth. However, the available bandwidth is capped by the BAND scheduler,

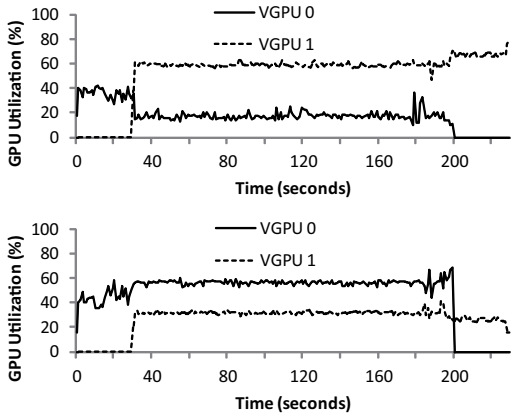


Figure 15: Util. of virtual GPUs with the MRQ scheme (upper for compute and lower for memory-copy).

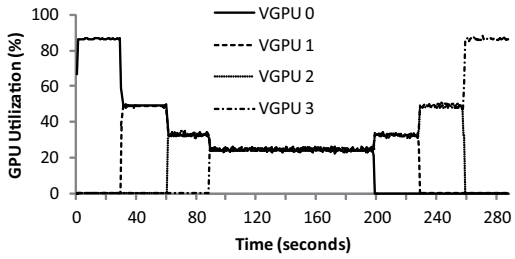


Figure 16: Util. of virtual GPUs under fair workloads.

i.e., both the compute and memory-copy operations are limited to about 50% of bandwidth at most. One can also observe that the MRQ scheme allowed the sum of compute and memory-copy bandwidth to exceed 100%.

We finally demonstrate the scalability of our virtual GPU support. Figure 16 shows the utilization of four virtual GPUs under the BAND scheduler, where all virtual GPUs execute four instances of the LUD benchmark task exhaustively to produce fair workloads. The workloads of each virtual GPU begin in turn at an interval of 30 seconds. Under such sane workloads, our virtual GPU support can provide fair bandwidth allocations, even if the system exhibits non-preemptive burst workloads.

8 Related Work

GPU Resource Management: TimeGraph [15] and GERM [2] provide a GPU command-driven scheduler integrated in the device driver. Specifically, TimeGraph achieves a prioritization and isolation scheme, extended with a resource sharing scheme [13] later on, whereas GERM enhances fair-share GPU resource control. Gdev also respects a prioritization, isolation, and fairness scheme, similar to TimeGraph and GERM, but adopts an API-driven scheduler model to reduce the number of scheduler invocations. The overhead of the command-driven scheduler model was also discussed in [15].

PTask [24] is an OS abstraction for GPU applications that optimizes data transfers between host and device memory using a data-flow programming model and a GPU scheduler. CGCM [11] is another solution based on the compiler and runtime library that dynamically and automatically optimizes the same sort of data trasfers. In contrast, Gdev does not support data-flow programming or automatic code generation. Alternatively, it provides programmers with an explicit API set to share device memory among GPU contexts. Such an IPC scheme can similarly reduce data transfer overhead.

RGEM [14] is a user-space runtime model for real-time GPGPU applications. It creates preemption points with data transfers between host and device memory in order to bound blocking times imposed on high-priority tasks. It also provides separate queues to demultiplex the schedulings of data transfers and computations. Albeit using a similar separate-queue scheme, Gdev addresses a core challenge of GPU resource management integreated in the OS to overcome the user-space limitations.

In addition to the aforementioned differences, Gdev can virtualize the GPU in the OS, which enables users to view a physical GPU as multiple logical GPUs for strong resource isolation. None of the previous work has also devided compute and memory bandwidth reservations, whereas Gdev accounts for these bandwidth reservations independently to maximize the overall GPU utilization. Furthermore, the previous work depend more or less on proprietary software or existing software stack, which could force design and implementation, if not concept, to adhere to user-space runtime libraries. Our prototype design and implementation of Gdev, in contrast, is fully self-contained, allowing the OS to fully control and even use GPUs as first-class computing resources.

GPUs as OS Resources: A significant limitation on the current GPU programming framework is that GPU applications must reside in the user space. KGPU [28] is a combination of the OS kernel module and user-space daemon, which allows the OS to use GPUs by up-calling the user-space daemon from the OS to access the GPU. On the other hand, Gdev provides OS modules with a set of traditional API functions for GPU programming, such as CUDA. Hence, a legacy GPU application program can execute in the OS, as it is, without any modifications and additional communications between the user space and the OS. In addition, we have shown that runtime support integrated in the OS is more reliable.

GPU Virtualization: VMGL [17] virtualizes GPUs at the OpenGL API level, and VMware's Virtual GPU [6] exhibits I/O virtualization through graphics runtimes. On the other hand, Pegasus [9] uses a hypervisor, Xen [1] in particular, to co-schedule GPUs and virtual CPUs in VMs. Nonetheless, these virtualization systems rely on user-space runtimes provided by proprietary software,

preventing the system from managing GPU resources in a fine-grained manner. In addition, they are mainly designed to make GPUs available in virtualized environments, but are not tailored to isolate GPU resources among users. Gdev provides virtual GPUs with strong time and space partitioning, and hence could underlie these GPU virtualization systems.

I/O Scheduling: GPU scheduling deals with a non-preemptive nature of execution as well as traditional I/O scheduling. Several disk bandwidth-aware schedulers [8, 22, 30], for example, contain a similar idea to the Gdev scheduler. Unlike typical I/O devices, however, GPUs are coprocessors operating asynchronously with own sets of execution contexts, registers, and memory. Therefore, Gdev adopts a scheduling algorithm more appropriate for compute-intensive workload.

Compile-Time and Application Approaches: GPU resources can also be managed by application programs without using drivers and libraries [4, 7, 26]. However, these approaches essentially need to modify or recompile the programs using specific compilers and/or algorithms. Thus, a generality of programming frameworks need to be compromised. In contrast, Gdev allows applications to use traditional GPU programming frameworks.

9 Conclusion

This paper has presented Gdev, a new approach to GPU resource management that integrates runtime support into the OS. This runtime-unified OS approach realizes new memory management and scheduling schemes that enable a wide class of applications to GPUs as first-class computing resources in general-purpose multi-tasking systems. We implemented a prototype system of Gdev, and conducted thorough experiments to demonstrate the advantage and disadvantage of using our Gdev approach. Our conclusion is that Gdev needs to compromise some basic performance due to incorporating runtime support in the OS, but can enhance GPU resource management for multi-tasking systems and allow the OS itself to use GPUs for computations.

Our prototype system and application programs used in the performance evaluation are all open-source, and may be downloaded from our website [25].

References

[1] 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 ACM Symposium on Operating Systems Principles* (2003).

[2] Bautin, M., Dwarakinath, A., and Chiueh, T. Graphics engine resource management. In *Proc. of Annual Multimedia Computing and Networking Conference* (2008).

[3] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J., Lee, S.-H., and Skadron, K. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. of IEEE International Conference on Workload Characterization* (2009), pp. 44–54.

[4] Chen, L., Villa, O., Krishnamoorthy, S., and Gao, G. Dynamic Load Balancing on Single- and Multi-GPU Systems. In *Proc. of IEEE International Parallel and Distributed Processing Symposium* (2010).

[5] Diamos, G., Kerr, A., Yalamanchili, S., and Clark, N. Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proc. of ACM International Conference on Parallel Architectures and Compilation Techniques* (2010), pp. 353–364.

[6] Dowty, M., and Sugeman, J. GPU virtualization on VMware's hosted I/O architecture. *ACM Operating Systems Review 43*, 3 (2009), 73–82.

[7] Guevara, M., Gregg, C., Hazelwood, K., and Skadron, K. Enabling Task Parallelism in the CUDA Scheduler. In *Proc. of Workshop on Programming Models for Emerging Architectures* (2009), pp. 69–76.

[8] Gulati, A., Ahmad, I., and Waldspurger, C. PARDA: Proportional allocation of resources for distributed storage access. In *Proc. of USENIX Conference on File and Storage Technology* (2009).

[9] Gupta, V., Schwan, K., Tolia, N., Talwar, V., and Ranganathan, P. Pegasus: Coordinated scheduling for virtualized accelerator-based systems. In *Proc. of USENIX Annual Technical Conference* (2011).

[10] Hand, S., Jang, K., Park, K., and Moon, S. PacketShader: a GPU-accelerated software router. In *Proc. of ACM SIGCOMM* (2010).

[11] Jablin, T., Prabhu, P., Jablin, J., Johnson, N., Beard, S., and August, D. Automatic CPU-GPU communication management and optimization. In *Proc. of ACM Conference on Programming Language Design and Implementation* (2011).

[12] Jang, K., Han, S., Han, S., Moon, S., and Park, K. SSLShader: Cheap SSL acceleration with commodity processors. In *Proc. of USENIX Conference on Networked Systems Design and Implementation* (2011).

[13] Kato, S., Lakshmanan, K., Ishikawa, Y., and Rajkumar, R. Resource sharing in GPU-accelerated windowing systems. In *Proc. of IEEE Real-Time and Embedded Technology and Aplications Symposium* (2011), pp. 191–200.

[14] Kato, S., Lakshmanan, K., Kumar, A., Kelkar, M., Ishikawa, Y., and Rajkumar, R. RGEM: A responsive GPGPU execution model for runtime engines. In *Proc. of IEEE Real-Time Systems Symposium* (2011), pp. 57–66.

[15] Kato, S., Lakshmanan, K., Rajkumar, R., and Ishikawa, Y. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proc. of USENIX Annual Technical Conference* (2011).

[16] Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A., Kaldewey, T., Lee, V., Brandt, S., and Dubey, P. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In *Proc. of ACM International Conference on Management of Data* (2010).

[17] Lagar-Cavilla, H., Tolia, N., Satyanarayanan, M., and de Lara, E. VMM-independent graphics acceleration. In *Proc. of ACM/Usenix International Conference on Virtual Execution Environments* (2007), pp. 33–43.

[18] Martin, K., Faith, R., Owen, J., and Akin, A. *Direct Rendering Infrastructure, Low-Level Design Document*. Precision Insight, Inc., 1999.

[19] McNaughton, M., Urmson, C., Dolan, J., and Lee, J.-W. Motion Planning for Autonomous Driving with a Conformal Spatiotemporal Lattice. In *Proc. of IEE International Conference on Robotics and Automation* (2011), pp. 4889–4895.

[20] NVIDIA. NVIDIA's next generation CUDA compute architecture: Fermi. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2009.

[21] NVIDIA. CUDA 4.0. `http://developer.nvidia.com/cuda-toolkit-40`, 2011.

[22] Povzner, A., Kaldewy, T., Brandt, S., Golding, R., Wong, T., and Maltzahn, C. Efficient guaranteed disk request scheduling with Fahrrad. In *Proc. of ACM European Conference on Computer Systems* (2008), pp. 13–25.

[23] Pronovost, S., Moreton, H., and Kelley, T. Windows Display Driver Model (WDDM v2 and beyond. Windows Hardware Engineering Conference, 2006.

[24] Rossbach, C., Currey, J., Silberstein, M., Ray, B., and Witchel, E. PTask: Operating system abstractions to manage GPUs as compute devices. In *Proc. of ACM Symposium on Operating Systems Principles* (2011).

[25] S. Kato. Gdev Project. `http://sys.ertl.jp/gdev/`, 2012.

[26] Saba, A., and Mangharam, R. Anytime Algorithms for GPU Architectures. In *Proc. of IEEE Real-Time Systems Symposium* (2011).

[27] Shimokawabe, T., Aoki, T., Muroi, C., Ishida, J., Kawano, K., Endo, T., Nukada, A., Maruyama, N., and Matsuoka, S. An 80-Fold Speedup, 15.0 TFlops, Full GPU Acceleration of Non-Hydrostatic Weather Model ASUCA Production Code. In *Proc. of ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis* (2010).

[28] Sun, W., Ricci, R., and Curry, M. GPUstore: Harnessing GPU Computing for Storage Systems in the OS Kernel.

[29] TOP500 Supercomputing Site. `http://www.top500.org/`, 2011.

[30] Wang, Y., and Merchant, A. Proportional-share scheduling for distributed storage systems. In *Proc. of USENIX Conference on File and Storage Technology* (2007).

Gnothi: Separating Data and Metadata for Efficient and Available Storage Replication

Yang Wang, Lorenzo Alvisi, and Mike Dahlin
The University of Texas at Austin
{*yangwang, lorenzo, dahlin*}*@cs.utexas.edu*

Abstract: This paper describes Gnothi, a block replication system that separates data from metadata to provide efficient and available storage replication. Separating data from metadata allows Gnothi to execute disk accesses on subsets of replicas while using fully replicated metadata to ensure that requests are executed correctly and to speed up recovery of slow or failed replicas.

Performance evaluation shows that Gnothi can achieve 40-64% higher write throughput than previous work and significantly save storage space. Furthermore, while a failed replica recovers, Gnothi can provide about 100-200% higher throughput, while still retaining the same recovery time and while guaranteeing that recovery eventually completes.

1 Introduction

An ideal storage system should provide good availability and durability, strong correctness guarantees, low cost, and fast failure recovery. Existing replicated storage systems make different trade-offs among these properties: 1) synchronous primary-backup systems [10, 13, 15] require $f+1$ replicas to tolerate f crash faults, but they risk data loss if there are timing errors; 2) asynchronous full replication systems [4, 5, 8, 18] use asynchronous agreement [20, 21] to ensure correctness despite timing errors, but send data to $2f+1$ replicas to tolerate f crash failures and thus have higher costs than synchronous primary-backup systems; 3) asynchronous partial replication systems [22, 29] still require $2f+1$ replicas, but they only activate $f+1$ of the replicas in the failure-free case; the spare replicas are activated only if some of the active ones fail. Although existing partial replication approaches are promising for replicating services with small amounts of state, they are not well-suited for replicating a block storage service because after a failure the system becomes unavailable until it activates a spare replica, which requires copying all of the state from available replicas. If the copying can be done at, say, 100MB/s, then the fail-over time would exceed 2.7 hours per terabyte of storage capacity.

This paper describes Gnothi[1], a new storage block system that achieves all these properties. Gnothi replicates data to guarantee availability and durability when replicas fail. To guarantee correctness despite timing errors, Gnothi uses $2f+1$ replicas to perform asynchronous state machine replication [20, 21, 27]. To reduce network bandwidth, disk arm overhead, and storage cost, Gnothi executes updates to different blocks on different subsets of replicas. The key challenge is to perform partial replication while not hurting availability or durability. Gnothi meets this challenge by using two key ideas.

First, to ensure availability during failure and recovery, Gnothi *separates data from metadata* so that metadata is replicated on all replicas while data for a given block is replicated only to a preferred subset for that block. A replica's metadata keeps the status of each block in the system, including whether the replica holds the block's current version. Replicating metadata to all replicas allows a replica to always process a request correctly, even while it is recovering after having missed some updates.

Second, to ensure durability during failures, Gnothi *reserves a small fraction (e.g. 10%) of storage on each replica* to buffer writes to unavailable replicas. While up to f of a block's *preferred replicas* are unresponsive, Gnothi buffers writes in the reserve storage of up to f of the block's available *reserve replicas*. Directing writes to a reserved replica when a block's preferred replica is unavailable guarantees that each new update is always written to $f+1$ replicas even if some replicas fail. Gnothi allows a tradeoff between availability and space cost: data is writeable in the face of f failures as long as failed nodes are repaired before the reserve space is exhausted. To guarantee write availability regardless of failure duration or repair time, conservative users can configure the system with the same space as asynchronous full replication ($2f+1$ actual storage blocks per logical block). Given that in Gnothi replicas recover quickly, analysis of several traces shows that a 10% reserve is enough to guarantee write availability for many workloads.

Gnothi combines those ideas to ensure availability and durability during failures and to make recovery fast despite partial replication. In summary, Gnothi provides the following guarantees: when an update completes, data is stored on $f+1$ disks; all reads and writes are lineariz-

[1] "Gnothi S'auton" (Γνῶθι σ'αυτόν is the ancient Greek aphorism "Know thyself".

able [17]; reads always return the most current data even though some replicas may have stale versions of some blocks; the system is available for reads as long as there are at most f failures; and the system is available for writes as long as there are at most f failures and failed replicas recover or are replaced before the reserve buffer is fully consumed by new updates.

We implement Gnothi by modifying the ZooKeeper server [18]. Gnothi provides a block store API, and it can be used like a disk: users can mount it as a block device and create and use a filesystem on it. We evaluate Gnothi's performance both in the common case and during failure recovery and compare it with Gaios, a state-of-the-art Paxos-based block replication system [5]. The evaluation shows that Gnothi's write throughput can be 40%-64% higher than our implementation of a Gaios-like system while retaining Gaios's excellent read scalability. We also find that for systems with large amounts of state, separating data and metadata significantly improves recovery compared to traditional state machine replication. Unlike standard Paxos-based systems, Gnothi ensures that a recovering replica will eventually catch up regardless of the rate that new requests are processed, and unlike previous partial replicated systems, Gnothi remains available even while large amounts of state are rebuilt on recovering replicas.

2 Design

2.1 Interface and Model

Gnothi targets disk storage systems within small clusters of tens of machines. Because linearizability is composable, it is possible to scale Gnothi by composing multiple small clusters, but this is not discussed or evaluated in this paper.

Gnothi provides an interface similar to a disk drive: there is a fixed number of blocks with the same size, and applications can read or write a whole block. Block size is configurable. Our experiments use sizes ranging from 4KB to 1MB, but smaller or larger sizes are possible.

Gnothi provides linearizable reads and writes across different clients. Furthermore, if a client has multiple outstanding requests, Gnothi can be configured so that they will be executed in the order they were issued.

Gnothi is designed to be safe under the asynchronous model. It makes no assumption about the maximum communication delay between nodes, and thus it is impossible to detect whether a node has failed or it is just slow. Gnothi provides the same guarantees as previous asynchronous RSMs: the system is always safe (all correct replicas process the same sequence of updates), but it is only live (the system guarantees progress) during periods when the network is available and message delivery is timely. Gnothi uses $2f+1$ replicas to tolerate f omission/crash failures. Commission/Byzantine failures are not considered.

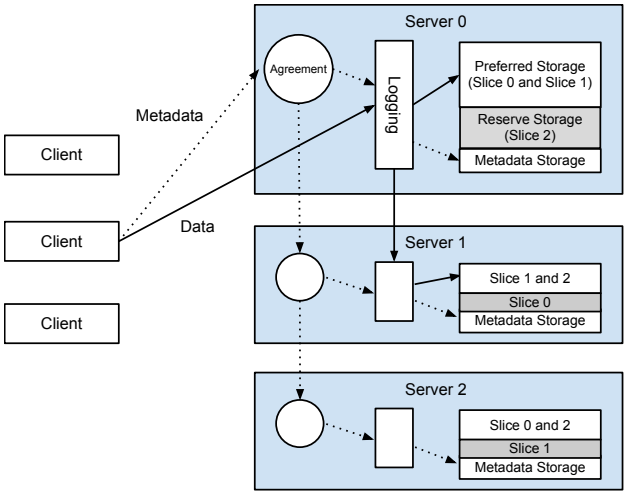


Figure 1: Data and metadata flow for a request to update a block in slice 1.

2.2 Architecture

As shown in Figure 1, Gnothi uses the Replicated State Machine (RSM) approach [27]: agreement modules on different replicas work together to guarantee that all replicas process the same client update requests in the same order. Requests are then logged and executed, and replies are sent to the client.

Gnothi splits metadata and data. Metadata is updated using state machine replication and is replicated at all $2f+1$ replicas, but data is replicated to just $f+1$ replicas. A replica marks a data block as *COMPLETE* or *INCOMPLETE* depending on whether or not the replica holds what it believes to be the block's current version.

- A block is *COMPLETE* at a replica if the replica stores a version of the block's data that corresponds to the latest update to the block recorded in that replica's metadata.
- A block is *INCOMPLETE* at a replica if the replica's metadata records a version of the block that is more recent than the latest data stored at the replica for that block.

Note that the concepts of *COMPLETE* and *INCOMPLETE* are different from those of *Fresh* and *Stale*. A block is *Fresh* if it contains the data of the latest update to that block and is *Stale* if it contains a previous version. In Gnothi, a *COMPLETE* block can be *Stale*. For example, this can happen when a node becomes disconnected and misses both the data and metadata update. Section 3.3 discusses how to avoid reading a *Stale* block.

When no failures or timeouts occur, Gnothi maps each block n to one of $2f+1$ *slices* and stores each slice

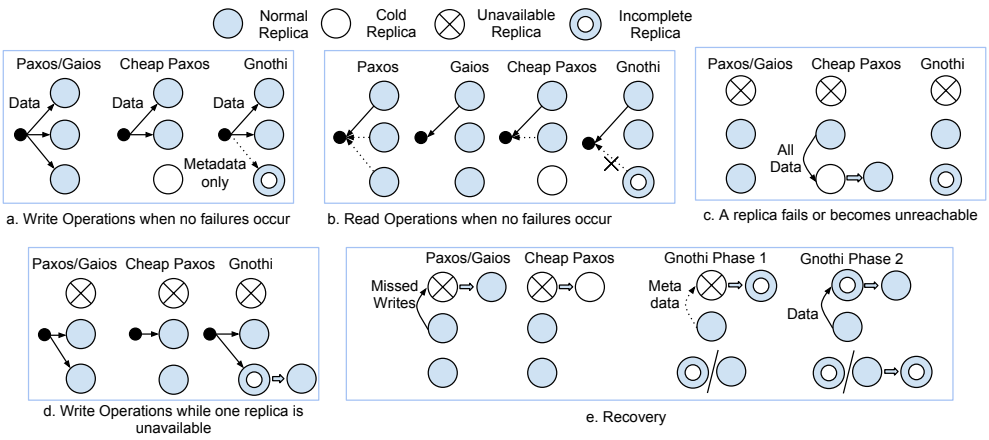


Figure 2: Gnothi protocols. We only show the logical flow of data and metadata in the figure, not actual messages. Consensus messages among servers are omitted, and we only show the flow and state for a single block. Multiple blocks are distributed among servers, so that each replica holds both *COMPLETE* and *INCOMPLETE* blocks.

on $f+1$ preferred replicas, from replica n to replica $(n-f)\%(2f+1)$. This ensures that the $2f+1$ slices are evenly distributed among different replicas, and that each replica is in the preferred quorum of $f+1$ different slices, which are the *PREFERRED* slices for that replica.

When failures or timeouts occur, a data block might be pushed to reserve storage on replicas out of its preferred quorum. We will show the detailed protocol later.

To simplify the description, we say that a block is *PREFERRED* at a replica if the replica is a member of the block's preferred quorum. Otherwise, we say that the block is *RESERVED* at that replica. We similarly say a request (read, write) is *PREFERRED*/*RESERVED* at a replica if it accesses a *PREFERRED*/*RESERVED* block at the replica.

Each replica allocates a preferred storage to store the data of *PREFERRED* writes, a reserve storage to store the data of *RESERVED* writes, and a relatively small metadata storage for each block's version and status.

2.3 Protocol Overview

This section presents an overview of Gnothi's protocol and compares Gnothi with the asynchronous full replication used by Paxos [20, 21], the asynchronous partial replication used by Cheap Paxos [22], and the state-of-the-art Paxos-based block replication system Gaios [5].

Figure 2.a shows a write operation when no failures occur. In Paxos and Gaios, a write operation is sent to, and executed on, all correct replicas. This seems redundant if our goal is to tolerate one failure: a natural idea is to send the write requests to two replicas first, and if they do not respond in time, try the third one [1]. Cheap Paxos adopts this idea by activating two replicas and leaving the other one as a cold backup [22, 29]. Gnothi incorporates a similar idea, but it still sends the metadata to the third replica, which executes the request by marking the corresponding data block as *INCOMPLETE*. Later, we will see that this metadata is critical to reducing the cost of failure and recovery.

Figure 2.b shows a read operation when no failures occur. In Paxos, the read is sent to all replicas and the client waits for two replies. The figure shows a common optimization that lets one replica send back the full reply and lets the others send back a hash or version number [9]. By using similar optimizations for its writes, Cheap Paxos executes the read on only two replicas. Gaios introduces a protocol that allows reads to execute on only one replica while still ensuring linearizability, and Gnothi uses Gaios's read protocol, with a slight modification to avoid reading *INCOMPLETE* blocks.

Figure 2.c shows what happens when one replica fails. Paxos and Gaios do not need special handling since the remaining two replicas hold all data. Cheap Paxos brings online the cold backup, which needs to fetch the data from the live replica: the system is unavailable until this transfer finishes, possibly for a long time if the system stores a large amount of data. In Gnothi, the third replica knows whether a block it stores is *COMPLETE* or not, so it can safely continue processing read requests by serving reads of *COMPLETE* blocks and redirecting reads of *INCOMPLETE* ones to the other replica. And it can also continue processing writes whose block belongs to the failed replica by storing data in its reserve storage. Therefore, Gnothi also does not need any special handling when a replica becomes unavailable.

Figure 2.d shows a write operation when a replica is unavailable. Paxos, Gaios, and Cheap Paxos do not need any special handling. For Gnothi, a replica may receive a *RESERVED* write and store it in its reserve storage to ensure that writes only complete when at least two nodes store their data. Read operations in this case are not different from those when no failure occurs.

Figure 2.e shows how recovery works. Paxos and Gaios both need to fetch all missing data before process-

ing new requests at the recovered replica. Cheap Paxos can just leave the recovered replica as the cold backup and does not need any special handling. Gnothi performs a two-phase recovery when a failed replica recovers.

In the first phase, the recovering replica fetches missing metadata from others. Since metadata is updated on all replicas, this phase of recovery proceeds as in a traditional RSM. After this phase is complete, the recovering replica can serve write requests even though full recovery is not complete yet: at this point the system stops consuming additional reserve storage on other replicas. Since the size of metadata is small, this phase is fast, and thus it is not necessary to allocate a large reserve storage.

In the second phase, the recovering replica rereplicates all missing or stale *PREFERRED* blocks. Gnothi performs this step asynchronously, so it can balance recovery bandwidth and execution bandwidth while still guaranteeing progress. Depending on the status of the recovering replica, there are two possible cases here: if all data on disk is lost, the recovering replica needs to rebuild its whole disk; if the data on disk is preserved, the recovering replica just needs to fetch the updates it missed during its failure. Note that Gnothi can continue processing reads and writes to all blocks during the second phase. If a node receives a read request for an *INCOMPLETE* block, it rejects the request, and the client retries with another replica.

2.4 Summary

Table 1 summarizes the costs of Gnothi and of previous work. In read cost, write cost, and space, Gnothi dominates Paxos, Gaios, and Cheap Paxos, improving on each in at least one dimension and approximating most in the others. For recovery and availability, Gnothi can perform the heavy data transfer in the background concurrently with serving new requests, while in Paxos and Gaios, the recovering replica must wait for the transfer to finish, and in Cheap Paxos, the whole system must halt until the transfer completes.

3 Detailed Design

This section presents in detail how Gnothi stores and accesses data and metadata, and how it performs recovery after a replica fails.

3.1 Data and Metadata

Gnothi splits the storage space into $2f+1$ slices, with each replica in the preferred quorum of $f+1$ slices. A replica stores the data of its $f+1$ *PREFERRED* slices in its preferred storage, and allocates space for f *RESERVED* slices in its reserve storage. When all replicas are available, blocks are always written to preferred storage, but when some replicas are not available, blocks are stored in the reserve storage of replicas outside the block's preferred quorums.

If the per-slice size of preferred and reserve storage are the same, then the system can remain available indefinitely even if f replicas fail, but at the cost of $2f+1$ physical blocks for each logical block. In Section 3.5, we will show that, given Gnothi's fast recovery, a much smaller reserve storage is likely to suffice for many workloads. For now, let us assume that preferred and reserve storage have the same per-slice size.

In processing updates, Gnothi separates data and metadata. The data is carried in a "PrepareData" message, while the corresponding metadata is carried in a "WriteData" message; we will detail the messages' format in the following subsections. A client first sends PrepareData; upon receiving the message, a replica first logs it to disk and then stores it in a buffer until it receives the corresponding WriteData and can perform the actual write. We call the buffer the "PrepareData buffer" in the following sections. To avoid overflowing a replica's PrepareData buffer, Gnothi sets an upper bound on how many outstanding PrepareData requests a single client can have. If a replica finds its PrepareData buffer for a client is full, it stops receiving messages from that client until the buffer has room. Once it knows that the PrepareData has been stored by enough replicas, the client proceeds to send the WriteData. Replicas run an agreement protocol to guarantee that WriteData messages are processed in the same order by all correct replicas.

Note that a PrepareData may never be consumed by a replica. For example, a client could fail after sending the PrepareData but before sending the WriteData. To garbage-collect unused PrepareDatas, a client includes a client sequence number with each PrepareData and WriteData it sends, and a replica discards an unused PrepareData if it receives a WriteData with a higher sequence number. When the client fails and recovers, it sends to all replicas a special "new epoch" command. Replicas process the new epoch command using the same agreement protocol used to order WriteData messages: hence, by the time replicas enter a new epoch, they have processed the same sequence of WriteData messages. Once the new epoch command completes, all replicas can discard all PrepareDatas in the previous epoch. Notice that if the failed client does not recover, the replica cannot discard unused PrepareDatas; in asynchronous replication, it is impossible to know whether a client has permanently failed or is just slow. If the cost of a few megabytes per permanently-failed client is too high, the system can rely on an administrator or on a very long timeout (say, 1 day) to detect the disconnected client and clear its buffer.

Gnothi keeps metadata for each block: an 8-byte version number assigned by agreement to identify the

Protocol	Write	Read	Space	Failure	Recovery (Disk survived)	Recovery (Disk replaced)
Paxos	2f+1	2f+1	2f+1	0	O(NB)	O(S)
Gaios	2f+1	1	2f+1	0	O(NB)	O(S)
Cheap Paxos	f+1	f+1	f+1+f (Cold)	O(S) (Blocking)	0	0
Gnothi	f+1	1	f+1+Δf $0 < \Delta \leq 1$	0	O(Nb)+O(NB)	O(Nb)+O($\frac{f+1}{2f+1}$S)

Table 1: Cost of Gnothi and previous work; S is the total storage space. N is the number of unique updated blocks missed by the recovering replica; B is the block size, and b is the metadata size for each block.

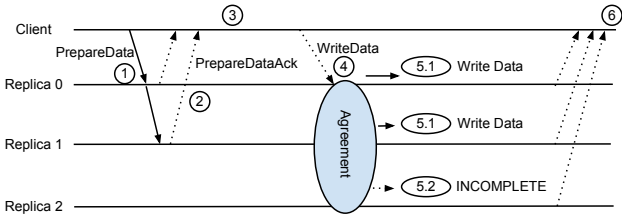


Figure 3: Write Protocol

block's last update, and an 8-byte requestID to connect the block to the PrepareData message. The version number and requestID are primarily used in failure recovery: we will show why Gnothi needs them and how to use them later. In addition, each replica keeps one bit for each block to identify whether or not the block is *COMPLETE* on the replica.

3.2 Write Protocol

The write protocol is illustrated in Figure 3:

(1) A client sends PrepareData(requestID, block data) to $f+1$ replicas, using the pair (clientID, clientSeqNo) to achieve a unique requestID. At first, the client targets the block's preferred quorum, and when a timeout occurs, the client tries other replicas. To prevent the client's network from becoming a bottleneck for sequential access we use chain replication [16, 26]: the client sends the data to one replica which forwards it to the next, and so on, in turn.

(2) A replica receiving the PrepareData puts it into a PrepareData buffer, logs it to disk, and sends an PrepareDataAck(requestID) to the client.

(3) The client waits for $f+1$ PrepareDataAcks. If there is a timeout, the client repeats Step (1), choosing some other replicas. When the network is available and message delivery is timely, this step is guaranteed to terminate as long as at least $f+1$ replicas are capable of processing requests.

(4) The client sends WriteData(requestID, block number) through the agreement protocol so that all replicas receive the same sequence of write commands. Gnothi uses code from ZooKeeper for agreement, but other Paxos-like protocols [5, 12] could be used.

(5) When receiving a WriteData message, a replica updates its metadata storage and tries to find the corresponding PrepareData in the PrepareData buffer by using the requestID as the identifier. There are three possible cases:

(5.1) The replica has both WriteData and PrepareData, and this is a *PREFERRED* write for the replica. The replica then writes the data to its preferred storage and marks the corresponding block as *COMPLETE*.

(5.2) The replica has WriteData but no PrepareData. The replica then marks the corresponding block as *INCOMPLETE*.

(5.3) The replica has both WriteData and PrepareData, and this is a *RESERVED* write for the replica. The replica then writes the data to the reserve storage and marks the corresponding block as *COMPLETE*. This case happens only when there are unavailable or slow replicas, so it is not shown in Figure 3.

In all cases, the replica sends a WriteAck(requestID) back to the client.

(6) The client waits for $f+1$ WriteAcks. If there is a timeout, the client repeats Step (4). If the WriteData has already been processed, the replicas send a WriteAck reply [9]; otherwise, they process the write request. Assuming there are at least $f+1$ functioning replicas, the client is guaranteed to get enough WriteAcks eventually.

To argue correctness, we observe that Step (3) guarantees that PrepareData is received by at least $f+1$ replicas and thus will not be lost; the agreement protocol guarantees that WriteData is eventually received by all correct replicas and thus will not be lost; and the agreement protocol provides the write linearizability guarantee. Notice that in Step (6), WriteAcks may not come from the same nodes that stored data and sent PrepareDataAcks in Step (2), but this is not a problem since the system is still protected against f failures. If one of the nodes storing data is slow, temporarily unavailable, or crashed but can recover locally, it will catch up with others using standard techniques [20, 21] and process the WriteData, so that the write will survive even if another node fails. Conversely, if a node permanently loses its data, the recovery protocol must restore full redundancy by fetching the failed node's state from the remaining replicas, but this case is no different whether the node that received the PrepareData and then permanently crashed did so before or after sending a WriteAck.

3.3 Read Protocol

For reads, we use the Gaios read protocol [5], modified slightly to handle *INCOMPLETE* blocks. (Steps (2)-(4) below are the same as described for Gaios):

(1) A client sends a Read (block number, replica ID) to the current agreement leader node, stating that it wants to

read that block from a specific target replica. Usually, the target is the first replica in the block's preferred quorum.

(2) The leader buffers the Read request and queries all other replicas: "Am I still the leader?"

(3) If the leader receives at least f "Yes" responses, it continues. Otherwise, it does nothing. This can happen if a slow replica still believes to be the leader, while enough other replicas have already moved on. In this case, the client will timeout, restart from Step (1), and try another replica as the leader.

(4) The leader attaches a version number to the Read and sends it to the target replica specified in the request. The version number is set to the number of write requests already proposed. This number is used later to ensure that the target replica does not read stale data.

(5) The target replica waits until the write with the specified version number is executed, and then it executes the Read. This synchronization prevents a slow replica from sending stale data to the client. There are two cases to consider:

(5.1) The corresponding block is *COMPLETE* : the target replica then sends the data to the client.

(5.2) The corresponding block is *INCOMPLETE*: the target replica sends an "INCOMPLETE" reply to the client. This allows the client to move to the next replica quickly, instead of waiting for a timeout.

(6) If the client receives the data, it finishes the Read. If it receives "INCOMPLETE" or times out, it chooses another replica and restarts from Step (1). The client chooses the target replica in round-robin fashion starting with the preferred quorum, so that all replicas will be tried.

When no failures or timeouts occur, Step (5.1) will always happen, since the client chooses a node from the preferred quorum as the target. When failures or timeouts occur, the client may try some other replicas, but during a period with timely message delivery, it will eventually succeed since some replica must hold the data.

Note that if the client issues a read and then a write to the same block before the read returns, the read can return the result of the later write. Gnothi assumes this is an acceptable behavior for block drivers [5], but a client can prevent it by blocking the later write when there is an outstanding read operation to the same block.

3.4 Failure and Recovery

Gnothi performs no special operations when replicas fail. A client may timeout in the read or write protocol and retry using some other replicas, or write data to some replicas not in the preferred quorum, which will store these *RESERVED* writes in their reserve storage.

Recovering a failed replica begins with replaying the replica's log. If the disk is damaged or the machine is entirely replaced, this step may fail but correctness is not affected. What cannot be recovered from the log is fetched from the other replicas in two phases: first to be restored is the metadata, and then any data missing from the failed replica's preferred slices. The recovering replica can process new requests once the first phase is complete, and it is fully recovered and no longer counts against our f threshold when the second phase is complete.

3.4.1 Phase 1: Metadata recovery

Gnothi replicates metadata on each node, so metadata recovery proceeds as it would in traditional RSMs: recovering replica sends to the primary the last version number it is aware of, to which the primary replies with a list of metadata records, if any, with higher version number. Besides the version number, each of these records includes a block number and a requestID.

For each received record, the replica then checks if it holds in its buffer a PrepareData with the same requestID: if so, it executes the write request and marks the block as *COMPLETE*. This check handles the case when a replica receives a PrepareData but fails before receiving the corresponding WriteData. In this case, the recovering replica should finish executing the write request, and the requestID is necessary to connect a PrepareData to its block. If there is no PrepareData in the buffer with the same requestID, the replica simply marks the block as *INCOMPLETE*.

When metadata recovery is complete, it is safe for the replica to process new requests, even though it may have some *INCOMPLETE* blocks. An update will overwrite the *INCOMPLETE* block, and a read will be redirected to other replicas with the *COMPLETE* block.

Gnothi transfers 24 bytes of metadata for each block during this phase. This is 6GB per terabyte of data using 4KB blocks and 24MB per terabyte for 1MB blocks, so the first phase typically takes a few seconds to a few minutes to complete. Note that during this metadata transfer, the other replicas continue to process new reads and writes.

3.4.2 Phase 2: Re-replicate

In the second phase, the recovering replica retrieves from the others the data for all the *INCOMPLETE* blocks in its preferred storage, thus freeing those replicas' reserve storage. If a replica retains its data on its local disk, it just needs to fetch the modified blocks. This case typically occurs when a replica crashes and recovers, becomes temporarily disconnected from the network, or becomes temporarily slow. If a replica loses its on-disk data as a result of a hardware fault, it needs to rebuild its storage by fetching all blocks in its slices' preferred storage.

This phase can take a long time, depending on the number of blocks to be fetched, but it is needed only to free the reserve space of other nodes, so that they are better equipped to mask future failures: once the replica re-

covers its metadata, it can process all writes to its slices, and it can process reads to the subset of blocks that are locally *COMPLETE*. Gnothi performs re-replication as a background task that can be throttled to balance the resources used for re-replication and for processing new client requests. Even if new client requests are processed at a high rate and re-replication proceeds at a low rate, re-replication will still eventually complete because the recovering replica's metadata allows it to process new requests while it is still catching up re-replicating missed old updates.

Every replica periodically checks its reserve storage: if a *RESERVED* block is *COMPLETE* on its preferred replicas, then the replica can safely delete the block from its reserve storage.

3.5 Reducing replication state

Each replica needs to reserve space for f *RESERVED* slices. It is always safe to set the size of reserve storage to be f times a slice size, so that it can absorb any number of writes to each slice. This approach amplifies storage costs by a factor of $2f+1$, since a data block is stored on a preferred quorum of $f+1$ replicas, and the other f replicas must reserve space for this block in the reserve storage. This means that when $f=1$ a replica must allocate one third of its storage space for reserve storage, and more when f is larger. This is the same space overhead as in the standard approach of Paxos or Gaios, which may be acceptable. When reducing replication costs is a concern, however, Gnothi also enables allocating less space for reserve storage. The risk of this thriftier approach is that if a failed replica does not recover or is not replaced soon, the reserve storage can fill, preventing the system from processing additional writes. However, filling the reserve storage does not put safety at risk, since data is always written to $f+1$ replicas. In general, Gnothi can allocate less space for reserve storage in any of the following cases: 1) the workload is read-heavy; 2) the workload is write-heavy but dominated by random writes so that the throughput is low; 3) the workload is write-heavy but has good locality. Our analysis of several disk traces suggests that, as long as the metadata is recovered quickly, allocating 10% of disk space as reserve storage is enough to guarantee write availiability for many workloads.

Specifically, we analyze two sets of traces from Microsoft: one is collected by Microsoft Research Cambridge [24] and it consists of 23 1-week disk traces under different workloads; the other is collected on Microsoft's production servers [19] and consists of 44 disk traces, whose lengths vary from 6 hours to 1 day. We choose these two sets of traces because they are recent and because they contain a variety of workloads including compiling, MSN Storage, SQL Server, computation, etc. We calculate the maximum usage ratio for each trace. To be precise, $MaxUsage(T)$ is the maximum number of different sectors written during any time interval of length T, divided by the total number of sectors.

In the Microsoft Cambridge Traces, only 2 of the 23 traces write to more than 10% of the disk space in a week. For the heaviest one, reserving 10% always allows at least 10 minutes to finish Phase 1 and recover all metadata before the system becomes unavailable to writes. A conservative administrator may reserve more for this workload.

In the Microsoft Production Server Traces, 38 of the 44 disk traces write to less than 10% of the space in their traces. For the heaviest one, reserving 10% always allows at least 10 minutes to complete Phase 1.

3.6 Metadata

Each replica stores both local and replicated metadata for every block. The local metadata consists of the *COMPLETE* bit for each block, and the replicated metadata includes the version number and requestID for each block.

In Gnothi, caching in memory the *COMPLETE* bit of each block is feasible in both size and cost. For example, with a small 4KB block each 1TB of disk storage requires about 30MB of *COMPLETE* bits. In May 2012, a commodity 2TB internal hard drive costs about $120 and a common 4GB memory DIMM costs about $25. This means that keeping *COMPLETE* bits in memory adds about 0.3% to the dollar cost of the disk data it tracks. Gnothi regularly stores checkpoints of the *COMPLETE* bits by writing the current state to local files.

The block number, version number, and requestID are 8 bytes each, and it would be costly for Gnothi to keep them all in memory. Gnothi uses a metadata storage design similar to that of BigTable [10, 23]. Each Gnothi node maintains in a local key-value store the mapping from logical block ID to version number and requestID. Metadata updates are logged to disk first as described before. Afterwards, to update a record, Gnothi first puts the record in a memory buffer; then when the buffer is full, Gnothi sorts the buffer according to the key and then writes the whole buffer to a new file. A background thread merges these files when there are too many of them. Metadata writes and merges are fast, since they are sequential writes to disk. Our micro benchmark shows that this approach can sustain a throughput of about 200K writes per second, which is enough for our needs. Reading from metadata storage only occurs when Gnothi recovers a crashed or slow replica by fetching metadata from another replica: this case requires a sequential scan of the metadata, which is again fast. Individual read operations do not access metadata storage, since a read operation only needs to access the *COMPLETE* bit.

4 Implementation

We implement Gnothi by modifying ZooKeeper's source code. In particular: 1) we reuse ZooKeeper's network and agreement modules to replicate metadata; 2) we add chain replication to forward data; 3) we modify the read protocol to provide linearizable and scalable reads; 4) we replace ZooKeeper's storage module with one that supports preferred, reserve, and metadata storage; 5) we modify ZooKeeper's logging system to record Prepare-Data messages; 6) we implement recovery as described in Section 3.4.

We apply several additional modifications to improve performance: first, Gnothi modifies ZooKeeper's agreement module to incorporate batching [9, 12], which improves performance by about 10% for the sequential write workload. Second, Gnothi reuses memory buffers to reduce memory allocation. ZooKeeper's server is implemented in Java, and our profiling shows that the overhead due to memory allocation and garbage collections is quite substantial, especially if the block size is large (ZooKeeper is explicitly not designed for large data blocks). To alleviate this problem, we reuse allocated memory buffers, which is not hard since they all have the same size. This device improves read performance by about 10-15% in our experiments.

5 Evaluation

5.1 Workload and Configuration

First, we evaluate the performance of Gnothi using micro benchmarks that issue both sequential and random reads and writes. We compare Gnothi's performance to a Gaios-like system that we implement (denoted in the following as G'), and to an unreplicated local disk. Both G' and Gnothi use the same code base; the only significant difference is that G' forwards all updates and stores all blocks at all replicas, while Gnothi processes each block at $f+1$ of the $2f+1$ replicas.

Second, we evaluate Gnothi in failure and recovery. We compare Gnothi with G' and Cheap Paxos in terms of availability, performance in the face of failures, and recovery time.

We use 5 machines as servers and 5 machines as clients. We run our performance evaluation experiments for two configurations: $f = 1$ (3 servers) and $f = 2$ (5 servers); we run the recovery experiments with $f = 1$. Gnothi's design calls for using a disk array for data storage and an additional disk to store log and metadata, but since our machines have only two Western Digital WD2502ABYS 250GB 7200 RPM hard drives, we evaluate Gnothi in a configuration where one disk is used as preferred and reserve storage, while the other stores the log and metadata. Each machine is equipped with a 4-core Intel Xeon X3220 2.40GHz CPU and 3GB of memory. For all experiments, we allocate 96GB of logical storage space replicated across nodes by the system under test. All machines are connected with 1Gbps ethernet.

For each experiment, we make sure there are enough client processes and outstanding requests to saturate the system; we make sure the experiment is long enough so that the write buffers are full; and we use the last 80% of requests to calculate the stable throughput. In all experiments, the read and write batches at each replica consist of, respectively, 100 and 10 requests. The values of other parameters (number of clients, number of outstanding requests per client, etc) depend on the block size (4K, 64K, 1M) and workloads (sequential/random write/read), and we do not list all of them. In general, sequential workloads and small blocks need more outstanding client requests to saturate the system; random workloads and big blocks need fewer; and random workloads with small blocks need a longer time to saturate the write buffer. For example, for the 4KB sequential write workloads, we use 30 clients, each with 200 outstanding requests, to saturate the system; for the 4KB random write workloads, 3 clients with 200 requests each are enough, but we need to run the experiments for 3 hours to measure the stable throughput; and for the 1MB sequential write workloads, it takes just 5 clients with 60 outstanding requests each to saturate the system.

5.2 I/O Throughput

Gnothi maximizes I/O throughput by executing reads and writes on subsets of disks.

Figure 4 shows the random I/O performance for $f = 1$ and $f = 2$. For random workloads, the bottleneck of the system is the seek time for each replica's data disk.

For write operations, Gnothi is 40-64% faster than writing to local disk or to G' for $f = 1$ and 53-75% for $f = 2$. Gnothi's advantage comes from only having to perform the writes at 2/3 (for $f = 1$) or 3/5 (for $f = 2$) of the nodes. As expected [5], the random write performance of G' is close to that of a single local disk because all replicas process all updates.

For read operations, Gnothi and G' perform identically since they use the same read protocol. Gnothi/G' is 2.5-3.4 times faster than a single local disk for $f = 1$ and 3.3-6.1 times faster for $f = 2$, because it executes each read on one replica. For small requests, the improvement factor can exceed $2f+1$ since each replica is responsible for $1/(2f+1)$ of the data, and thus the average seek time is reduced.

Note that for small random I/O, the local per-disk write bandwidth significantly exceeds the corresponding read bandwidth. The reason is that, once writes are committed to the log, we can buffer large numbers of writes

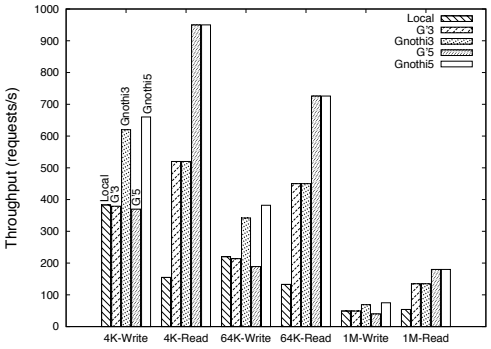


Figure 4: Random I/O with 3 (f=1) and 5 (f=2) servers.

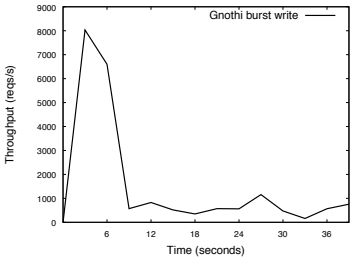


Figure 5: Burst writes. In the default configuration, Linux starts to flush dirty data to disk if 10% of total system memory pages are dirty.

before writing them back to the data disk, allowing the disk scheduler more opportunities to minimize seek and rotational latency. Reads, on the other hand, must be processed immediately, so the scheduler has fewer opportunities for optimization. Taking for example the 4KB random workload, a local disk can process 383 random writes per second, while it can only process about 155 random reads per second if there are 100 concurrent read requests.

Figure 5 shows the effect of a burst of random writes when $f = 1$ and the system buffers are not full. During the first few seconds, since writes are logged to the logging disk, buffered in memory, but not bottlenecked by flushing to the data disk, Gnothi's throughput is much higher than that of the data disk write back. Then, when the operating system detects that more than 10% of the system memory is dirty, it begins to write back data to disk at the same rate it receives new requests, and Gnothi slows down. Figure 4 shows the stable write throughput, where, to eliminate the effects of the initial spike, we run our experiments for sufficiently long (more than 3 hours) and calculate the throughput of the last 80% of requests.

Figure 6 shows the sequential I/O performance with $f = 1$ and $f = 2$.

For the sequential write workload with $f = 1$, Gnothi can achieve about 60MB/s with a 4KB block size and about 90MB/s with a 1MB block size. The bottleneck for 4KB block size is probably ZooKeeper's agreement, which is processing about 15K updates per second. For 1MB requests, our profiler shows that the bottleneck is

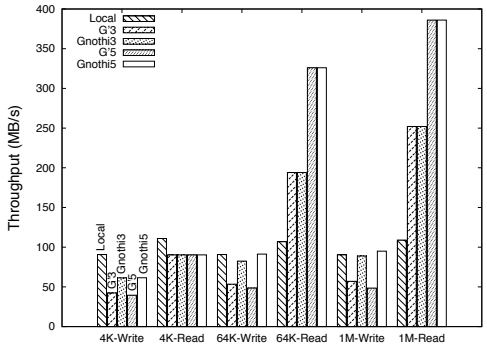


Figure 6: Sequential I/O with 3 (f=1) and 5 (f=2) servers.

probably Java's memory allocation and garbage collection, so customized memory management or a C implementation may achieve better performance. Compared to G', Gnothi is 44% to 56% faster because Gnothi directs writes to subsets of nodes.

For the read workload, Gnothi/G' can achieve a total bandwidth of about 250MB/s with 1MB blocks. One problem with reads is that if we use only 1 client, the client network becomes a bottleneck, and if we use multiple clients, then the workload is not fully sequential. This problem is more severe for small requests.

Compared with the $f = 1$ case, Gnothi's throughput for 64K and 1MB writes increases by about 10% when $f = 2$. In the 4KB case the bottleneck is agreement, so there is almost no improvement. G's throughput slightly decreases since its replication cost is higher. For reads, Gnothi/G' scales throughput by nearly a factor of 4 compared to a single disk.

5.3 Failure Recovery

Gnothi does three things to maximize availability and recovery speed. First, it fully replicates metadata, allowing the system to remain continuosly available in the face of up to f failures despite partial replication of data. Second, partial replication of data reduces recovery time, because the recovering node only needs to fetch $(f+1)/(2f+1)$ (e.g. 2/3 for $f = 1$) of the data. It also improves performance during recovery, because once metadata is restored, full block updates are only sent to and executed on the block's preferred quorum. Third, separation of data and metadata improves system throughput during recovery and reduces recovery time. The recovering node can catch up with other nodes even if they continue to process new updates at a high rate. In particular, since processing metadata is faster than processing full requests, Phase 1 of recovery can always catch up with missed and new requests. Once Phase 1 is complete, the recovering replica no longer falls behind as new requests are executed since it can process and store all new block updates directed to it, while it fetches old update bodies for all *INCOMPLETE* blocks

in its preferred slices.

Figures 7 and 8 look at two recovery scenarios. Figure 7 shows the case when a node temporarily fails and then recovers by fetching just the updated blocks it missed. Figure 8 shows the case when a node permanently fails and is replaced by a new node that must fetch all data from others. We run both experiments with $f = 1$, 4KB blocks, and a sequential write workload. We choose the sequential write workload because it is the most challenging workload for recovery, since during recovery the clients are writing new contents at a high speed, which consumes a large portion of the network and disk bandwidth from the servers.

In Figure 7, we kill one server 60 seconds after the experiment starts and restart it 60 seconds later. Here both Gnothi and G' suffer a brief drop in throughput while they wait for timeout and then continue without the failed node as a result of chain replication. After the replica restarts at time 120, it takes about 110 seconds (to time 230) to recover from its local disk (mainly replaying logs), and about 22 seconds (to time 252) to join the agreement protocol. Then Gnothi spends 26 seconds (to time 278) in Phase 1, during which the recovering replica fetches write metadata (but not data) and marks all updated blocks as *INCOMPLETE*. Once Phase 1 completes, the recovering replica begins servicing new requests, writing new writes to its local state, and marking updated blocks as *COMPLETE*. After Phase 1 completes, the recovering replica also begins Phase 2 of recovery by fetching from other replicas *INCOMPLETE* blocks in its preferred slices. Phase 2 completes at time 530, at which point recovery is complete, and Gnothi returns to its original throughput.

G's throughput starts at 50MB/s and remains the same while the failure occurs. After the replica resumes operation, in order to complete the recovery at time 530, it must throttle the rate at which it services new requests to about 16 MB/s.

Cheap Paxos is unavailable from time 30 to 230, since there is only one available replica and since it does not have sufficient time to copy 96GB to a spare machine. When the replica resumes operation, Cheap Paxos can immediately go back to normal (time 230) since it does not process any new requests during the failure period.

In Figure 8, one server is killed 300 seconds after the experiment is started and is replaced 300 seconds later by a new server whose local disk is initialized and needs to be fully rebuilt. Gnothi takes about 80 seconds in Phase 1 to fetch metadata from the primary. After Phase 1 completes, the recovering replica begins servicing new requests, and at the same time, re-replicating its disk by fetching blocks from others. The recovering replica completes re-replication at time 3400, and during this period, it can service new requests at a rate of about 48 MB/s.

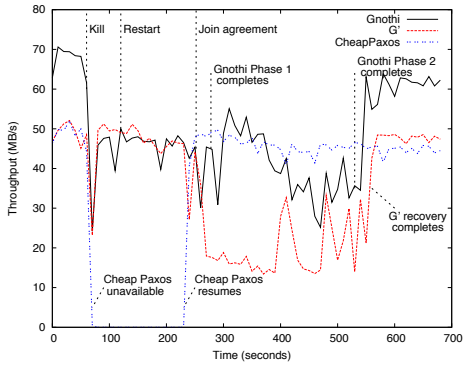


Figure 7: Failure recovery (catch up).

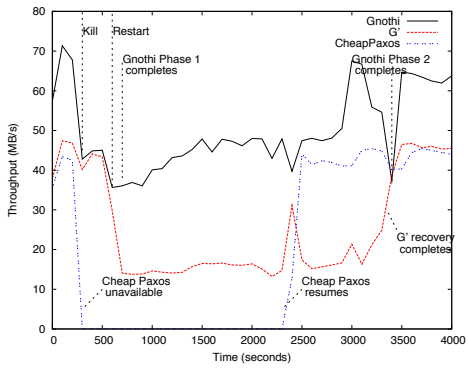


Figure 8: Failure recovery (re-replicate).

G' can also complete recovery at time 3400, but during this period, it can only service new requests at a rate of about 16 MB/s.

Cheap Paxos is unavailable before the re-replication is complete, but since it uses all its bandwidth to perform recovery, it can complete re-replication at time 2400.

Comparing Figure 7 and Figure 8, Gnothi's catch-up recovery takes less time than full re-replication (410 seconds vs 2800 seconds), but catch-up inflicts a bigger hit on throughput because when re-replicating all blocks, the disk accesses are always sequential, and when re-replicating a subset of them, the disk accesses may be random. This means the recovery cost per block is smaller in full re-replication, though the total number of blocks to be fetched is larger and this results in a higher client throughput but longer recovery time for full replication.

Both Gnothi and G' can tune a parameter to divide resources between servicing new requests and fetching state for recovery. The parameter is the time interval (ms) for a replica to issue a 16 MB state fetch request, where a smaller number means more aggressive recovery. In Figures 7 and 8, we configure this parameter so that Gnothi and G' can recover in similar time, while still providing reasonable throughput for new requests. In Figures 9 and 10, we show the effect of different configurations.

In Figure 9, we can see that Gnothi can always catch up, so the administrator can tune this parameter to bal-

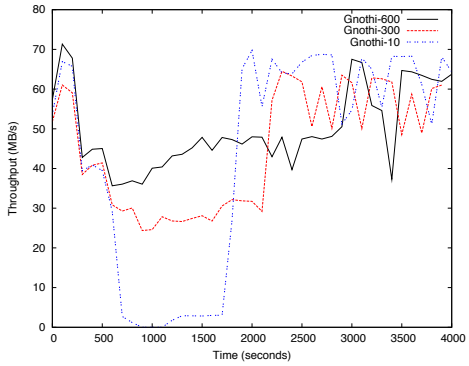


Figure 9: Gnothi with different recovery values.

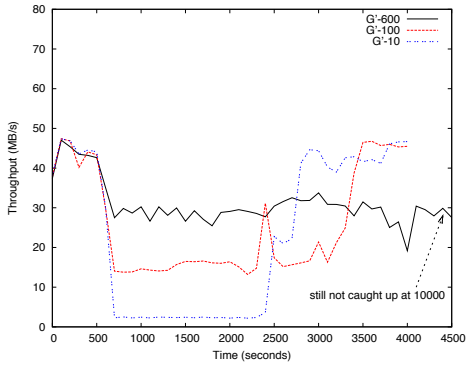


Figure 10: G' with different recovery values.

ance resources used for recovery and for processing new requests. Conversely, if G' sets this parameter too high (not aggressive), the recovering replica never catches up. For example, in Figure 10, the replica in the experiment with parameter 600 does not catch up, since the recovery speed is similar to the speed of processing new requests. Gnothi is almost always better than G' in our experiments: if recovery times are similar, Gnothi can provide better throughput during recovery; and if throughput is similar, Gnothi can recover faster.

6 Related Work

Replication techniques used to tolerate omission failures can be classified as either synchronous or asynchronous.

In synchronous replication [6, 7, 13], a primary replica provides the service to the clients, and if the primary replica fails, a backup replica takes over and continues to provide service. It takes $f+1$ replicas to tolerate f crash failures. There are three main disadvantages to synchronous primary backup [5]: 1) its correctness is not guaranteed when there are timing errors caused by network partitions or server overloading, since these faults can cause replicas to diverge; 2) to minimize correctness issues, the system must be configured with conservative timeouts that can hurt availability; 3) read throughput is limited by the capability of a single machine, since only the primary replica processes requests.

Asynchronous replication does not assume an upper bound on network latency or node response time, and hence can ensure correctness even in the face of relatively rare events like server overload, network overload, or network partitions. The traditional approach to asynchronous replication involves a Replicated State Machine (RSM), in which a consensus protocol guarantees that each correct replica receives the same sequence of requests and in which each replica is a deterministic state machine.

Paxos [20, 21] is representative of the asynchronous RSM approach, which requires $2f+1$ replicas to tolerate f crash failures. Paxos guarantees safety (all correct replicas receive the same sequence of requests) at all times and guarantees liveness (the system can make progress) when the network is available and node actions and message delivery are timely. Paxos uses timeouts internally, but it does not depend on their accuracy for safety and can adjust timeouts dynamically for liveness.

The standard Paxos protocol executes every request on each of the $2f+1$ replicas, with costs (in bandwidth, storage space, etc.) higher than synchronous replication. Much work has been done to reduce the cost of Paxos: Gaios does not log reads, executes them on only one replica, and nonetheless guarantees linearizabilty by adding new messages to the original Paxos protocol [5]. ZooKeeper [18] includes a fast read protocol that executes on a single replica, but it does not provide Paxos's linearizability guarantee.

On-demand instantiation (ODI) [22, 29] reduces write costs by executing requests on a preferred quorum of $f+1$ replicas. If one of the active replica fails, a backup replica is activated, but before it can start processing any request it must be initialized by fetching the current value of all replicated state. In storage systems with large amounts of data, this approach does not scale, as the system can be unavailable for hours while it transfers terabytes of data. Distler et al. [14] propose to alleviate this problem by replaying a per-object log on demand, but again this approach is not appropriate for replicating applications with large amounts of state, because its logs and snapshots are on a per-object basis; to reduce overhead, per-object garbage collection is performed infrequently, once every 100 updates, which means that the system stores 100 copies of each object at each replica.

Separating data and metadata Paris et al. [25] reduce the storage overhead of voting using volatile witnesses. Yin et al. [30] separate agreement from execution to reduce the number of execution nodes required for Byzantine replication, and Clement et al. [12] refine these techniques, but these separation techniques exploit a type of redundancy fundamentally different than that exploited by Gnothi. In particular, they exploit the redundancy

between tolerating Byzantine and omission faults; if u ("up") is the number of failures tolerated while ensuring liveness and r ("right") is the number tolerated while ensuring safety, then execution must be replicated to at least $u+max\{u,r\}+1$ nodes [11]. Gnothi "breaks" this bound by waiting for state updates to be successfully stored on $f+1$ of $2f+1$ nodes ($u+1$ of $2u+1$ in UpRight) and by using metadata to identify which nodes completed the last writes to which objects.

Several scalable cluster file systems [2, 3, 15, 28] are architected to separate data and metadata to allow one or more metadata managers to coordinate access to large numbers of storage servers by tracing where each object is stored. Gnothi, instead, focuses on scaling Paxos-based replication and providing strong consistency (linearizability) for arbitrary read/write workloads, and therefore maintains different types of metadata (block versions and requestID rather than mappings of objects/locations).

7 Conclusion

Gnothi is an asynchronous replicated storage system with low replication cost and fast failure recovery. Gnothi accomplishes this by separating data and metadata and replicating metadata on all replicas, while replicating data on subsets of them.

Gnothi demonstrates that full replication of metadata can 1) ensure that the system works correctly despite partial replication of data and 2) speed up recovery when replicas fail. The evaluation shows that Gnothi achieves higher throughput and availability than previous work.

Acknowledgement

We thank Manos Kapritsos for his unique blend of Σοφία and Φρόνησις. We also thank our shepherd, Jon Howell, and the anonymous reviewers for their insightful comments. This work was supported by NSF grant NSF-CiC-FRCC-1048269.

References

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-Scalable Byzantine Fault-Tolerant Services. In *SOSP*, 2005.

[2] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *OSDI*, 2002.

[3] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. *ACM Trans. Comput. Syst.*, 14(1), 1996.

[4] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR*, 2011.

[5] W. Bolosky, D. Bradshaw, R. Haagens, N. Kusters, and P. Li. Paxos Replicated State Machines as the Basis of a High-Performance Data Store. In *NSDI*, 2011.

[6] T. C. Bressoud and F. B. Schneider. Hypervisor-based Fault-tolerance. *ACM Trans. Comput. Syst.*, 14(1), 1996.

[7] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. Primary-Backup Protocols: Lower Bounds and Optimal Implementations. In *CDCCA*, 1992.

[8] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[9] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Trans. Comput. Syst.*, 20(4), 2002.

[10] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, 2006.

[11] A. Clement. *UpRight Fault Tolerance*. PhD thesis, 2010.

[12] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. UpRight Cluster Services. In *SOSP*, 2009.

[13] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *NSDI*, 2008.

[14] T. Distler and R. Kapitza. Increasing Performance in Byzantine Fault-Tolerant Systems with On-Demand Replica Consistency. In *Eurosys*, 2011.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *SOSP*, 2003.

[16] R. Guerraoui, R. Levy, B. Pochon, and V. Quema. Throughput Optimal Total Order Broadcast for Cluster Environments. *ACM Trans. Comput. Syst.*, 28(2), 2010.

[17] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990.

[18] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIX*, 2010.

[19] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda. Characterization of Storage Workload Traces from Production Windows Servers. In *IISWC*, 2008.

[20] L. Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2), 1998.

[21] L. Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4), 2001.

[22] L. Lamport and M. Masa. Cheap Paxos. In *DSN*, 2004.

[23] M. Mammarella, S. Hovsepian, and E. Kohler. Modular Data Storage with Anvil. In *SOSP*, 2009.

[24] D. Narayanan, A. Donnelly, and A. Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. In *FAST*, 2008.

[25] J.-F. Paris and D. Long. Voting with Regenerable Volatile Witnesses. In *ICDE*, 1991.

[26] R. Renesse and F. Schneider. Chain Replication for Supporting High Throughput and Availability. In *OSDI*, 2004.

[27] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4), 1990.

[28] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *MSST*, 2010.

[29] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet. ZZ and the Art of Practical BFT Execution. In *Eurosys*, 2011.

[30] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *SOSP*, 2003.

Dynamic Reconfiguration of Primary/Backup Clusters

Alexander Shraer *Benjamin Reed*
Yahoo! Research
{*shralex, breed*}*@yahoo-inc.com*

Dahlia Malkhi
Microsoft Research
dalia@microsoft.com

Flavio Junqueira
Yahoo! Research
fpj@yahoo-inc.com

Abstract

Dynamically changing (reconfiguring) the membership of a replicated distributed system while preserving data consistency and system availability is a challenging problem. In this paper, we show that reconfiguration can be simplified by taking advantage of certain properties commonly provided by Primary/Backup systems. We describe a new reconfiguration protocol, recently implemented in Apache Zookeeper. It fully automates configuration changes and minimizes any interruption in service to clients while maintaining data consistency. By leveraging the properties already provided by Zookeeper our protocol is considerably simpler than state of the art.

1 Introduction

The ability to reconfigure systems is critical to cope with the dynamics of deployed applications. Servers permanently crash or become obsolete, user load fluctuates over time, new features impose different constraints; these are all reasons to reconfigure an application to use a different group of servers, and to shift roles and balance within a service. We refer to this ability of a system to dynamically adapt to a changing set of machines or processes as *elasticity*.

Cloud computing has intensified the need for elastic long lived distributed systems. For example, some applications such as sports and shopping are seasonal with heavy workload bursts during championship games or peak shopping days. Such workloads mean that elasticity is not a matter of slowly growing a cluster; it may mean that a cluster grows by an order of magnitude only to shrink by the same order of magnitude shortly after.

Unfortunately, at the back-end of today's cloud services, one frequently finds a coordination service which itself is not elastic, such as ZooKeeper [12]. Companies such as Facebook, LinkedIn, Netflix, Twitter, Yahoo!, and many others, use Zookeeper to track failures and configuration changes of distributed applications; application developers just need to react to events sent to them by the coordination service. However, Zookeeper users have been asking repeatedly since 2008 to facilitate reconfiguration of the service itself, and thus far, the road to elasticity has been error prone and hazardous: Presently, servers cannot be added to or removed from a running ZooKeeper cluster and similarly no other configuration parameter (such as server roles, network addresses and ports, or the quorum system) can be changed dynamically. A cluster can be taken down, reconfigured, and restarted, but (as we explain further in Section 2) this process is manually intensive, error prone and hard to execute correctly even for expert ZooKeeper users. Data corruption and split-brain[1] caused by misconfiguration of Zookeeper has happened in production[2]. In fact, configuration errors are a primary cause of failures in production systems [22]. Furthermore, service interruptions are currently inevitable during reconfigurations. These negative side-effects cause operators to avoid reconfigurations as much as possible. In fact, operators often prefer to over-provision a Zookeeper cluster than to reconfigure it with changing load. Over-provisioning (such as adding many more replicas) wastes resources and adds to the management overhead.

Our work provides a reconfiguration capability using ZooKeeper as our primary case-study. Our experience with ZooKeeper in production over the past years has lead us to the following requirements: first, ZooKeeper is a mature product that we do not want to destabilize; a solution to the dynamic reconfiguration problem should not require major changes, such as limiting concurrency or introducing additional system components. Second, as many Zookeeper-based systems are online, service disruptions during a reconfiguration should be minimized and happen only in rare circumstances. Third, even if there are failures during reconfiguration, data integrity, consistency or service availability must not be compromised, for instance, split-brain or loss of service due to partial configuration propagation should never be possible. Finally, we must support a vast number of clients who seamlessly migrate between configurations.

We use the Zookeeper service itself for reconfiguration, but we ruled out several straw-man approaches. First, we could have used an external coordination ser-

[1] In a *split-brain* scenario, servers form multiple groups, each independently processing client requests, hence causing contradictory state changes to occur.

[2] `http://search-hadoop.com/m/ek5ej2dOQsB`

vice, such as another ZooKeeper cluster, to coordinate the reconfiguration, but this would simply push the reconfiguration problems to another system and add extra management complexity. Another naïve solution would be to store configuration information as a replicated object in Zookeeper. When a ZooKeeper server instance comes up, it looks at its replica of the state to obtain the configuration from the designated object. While this solution is simple and elegant, it is prone to inconsistencies. Some replicas may be behind others, which means they could have different configuration states. In a fixed configuration, a consistent view of the system can be obtained by contacting a quorum of the servers. A reconfiguration, however, changes the set of servers and therefore guaranteeing a consistent view requires additional care. Consequently, reading the configuration from an object in Zookeeper may lead to unavailability or, even worse, corrupt data and split-brain.

Indeed, dynamically reconfiguring a replicated distributed system while preserving data consistency and system availability is a challenging problem. We found, however, that high-level properties provided by Zookeeper simplify this task. Specifically, ZooKeeper employs a primary/backup replication scheme where a single dynamically elected primary executes all operations that change the state of the service and broadcasts state-updates to backups. This method of operation requires that replicas apply state changes according to the order of primaries over time, guaranteeing a property called *primary order* [13]. Interestingly, this property is preserved by many other primary/backup systems, such as Chubby [5], GFS [8], Boxwood [19], PacificA [21] and Chain-Replication [20] (see Section 6). These systems, however, resort to an external service for reconfiguration. In this work we show that leveraging primary order simplifies reconfiguration. By exploiting primary order we are able to implement reconfiguration without using an external service and with minimal changes to ZooKeeper (in fact, reconfigurations are pipelined with other operations and treated similarly) while guaranteeing minimal disruption to the operation of a running system. We believe that our methods may be applied to efficiently reconfigure any Primary/Backup system satisfying primary order.

Previous reconfiguration approaches, such as the one proposed by Lamport [15], may violate primary order, cause service disruption during reconfiguration, as well as impose a bound on the concurrent processing of *all* operations due to uncertainty created by the ability to reconfigure (see Section 2). Similar to our approach, FRAPPE [4] imposes no such bounds, but requires rollback support and complex management of speculative execution paths, not needed in our solution.

Our reconfiguration protocol also encompasses the clients. As the service configuration changes, clients should stay connected to the service. Literature rarely mentions the client side of reconfiguration, usually stating the need for a name-service (such as DNS), which is of course necessary. However, its also crucial to re-balance client connections across new configuration servers and at the same time prevent unnecessary client migration which may overload servers, severely degrading performance. We propose a probabilistic load-balancing scheme to move as few clients as possible and still maintain an even distribution of clients across servers. When clients detect a change, they each apply a migration policy in a distributed fashion to decide whether to move to a new server, and if so, which server they should move to.

In summary, this paper makes the following contributions:

- An observation that primary order allows for simple and efficient dynamic reconfiguration.
- A new reconfiguration protocol for Primary/Backup replication systems preserving primary order. Unlike all previous reconfiguration protocols, our new algorithm does not limit concurrency, does not require client operations to be stopped during reconfigurations, and does not incur a complicated management overhead or any added complexity to normal client operation.
- A decentralized, client-driven protocol that rebalances client connections across servers in the presence of service reconfiguration. The protocol achieves a proven uniform distribution of clients across servers while minimizing client migration.
- Implementation of our reconfiguration and load-balancing protocols in Zookeeper (being contributed to Zookeeper codebase) and analysis of their performance.

2 Background

This section provides the necessary background on ZooKeeper, its way of implementing the primary/backup approach, and the challenges of reconfiguration.

Zookeeper. Zookeeper totally orders all writes to its database. In addition, to enable some of the most common use-cases, it executes requests of every client in FIFO order. Zookeeper uses a primary/backup scheme in which the primary executes all write operations and broadcasts state changes to the backups using an atomic broadcast protocol called Zab [13]. ZooKeeper replicas process read requests locally. Figure 1 shows a write operation received by a primary. The primary executes the write and broadcasts a state change that corresponds to the result of the execution to the backups. Zab uses quo-

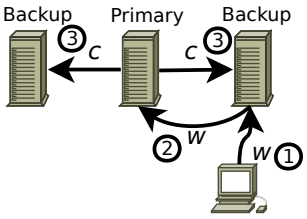


Figure 1: The processing of a write request by a primary. 1. a backup receives the request, w; 2. the backup forwards w to the primary; 3. the primary broadcasts the new state change, c, that resulted from the execution of w.

rums to commit state changes. As long as a quorum of servers are available, Zab can broadcast messages and ZooKeeper remains available.

Primary/Backup replication a la Zab. Zab is very similar to Paxos [15], with one crucial difference – the agreement is reached on full history prefixes rather than on individual operations. This difference allows Zab to preserve primary order, which may be violated by Paxos (as shown in [13]). We now present an overview of the protocol executed by the primary. Note that the protocol in this section is abstract and excludes many details irrelevant to this paper. The protocol has two parts, each involving an interaction with a quorum: A startup procedure, which is performed only once, and through which a new leader[3] determines the latest state of the system[4]; and a steady-state procedure for committing updates, which is executed in a loop.

Zab refers to the period of time that a leader is active as an *epoch*. Because there is only one leader active at a time, these epochs form a sequence, and each new epoch can be assigned a monotonically increasing integer called the epoch number. Specifically, each backup maintains two epoch identifiers: the highest epoch that it received from any primary in a startup phase, $e_{prepare}$, and the highest epoch of a primary whose history it adopted in steady-state, e_{accept}.

Startup: A candidate leader b chooses a *unique* epoch e and sends a PREPARE message to the backups. A backup receiving a PREPARE message acts as follows:

- If $e \geq e_{prepare}$, it records the newly seen epoch by setting $e_{prepare}$ to e and then responds with an ACK message back to the candidate.
- The ACK includes a history prefix H consisting of state-updates previously acknowledged by the backup, as well as the epoch e_{accept}.

When b collects a quorum of ACK messages, it adopts a history H received with the highest e_{accept} value, breaking ties by preferring a longer H.

Steady-state: For every client request op, the primary b applies op to its update history H and sends an ACCEPT message to the backups containing e and the adopted history H; in practice, only a delta-increment of H is sent each time. When a backup receives an ACCEPT message, if $e \geq e_{prepare}$, it adopts H and sets both $e_{prepare}$ and e_{accept} to e. It then sends an acknowledgment back to b. Once a quorum of followers have acknowledged the ACCEPT message, and hence the history prefix, b commits it by sending a COMMIT message to the backups.

Primary order. Because the primary server broadcasts state changes, Zab must ensure that they are received in order. Specifically, if state change c is received by a backup from a primary, all changes that precede c from that primary must also have been received by the backup. Zab refers to this ordering guarantee as *local primary order*. The local primary order property, however, is not sufficient to guarantee order when primaries can crash. It is also necessary that a new primary replacing a previous primary guarantees that once it broadcasts new updates, it has received all changes of previous primaries that have been delivered or that will be delivered. The new primary must guarantee that no state changes from previous primaries succeed its own state changes in the order of delivered state changes. Zab refers to this ordering guarantee as *global primary order*.

The term *primary order* refers to an ordering that satisfies both local and global primary orders. While the discussion above has been in the context of ZooKeeper and Zab, any primary/backup system in which a primary executes operations and broadcasts state changes to backups will need primary order. The importance of this property has already been highlighted in [13, 3]. Here, we further exploit this property to simplify system reconfiguration.

Configurations in Zookeeper. A ZooKeeper deployment currently uses a static configuration S for both clients and servers, which comprises a set of servers, with network address information, and a quorum system. Each server can be defined as a *participant*, in which case it participates in Zab as a primary or as backup, or an *observer*, which means that it does not participate in Zab and only learns of state updates once they are committed. For consistent operation each server needs to have the same configuration S, and clients need to have a configuration that includes some subset of S.

Performing changes to a ZooKeeper configuration is currently a tricky task. Suppose, for example, that we are to add three new servers to a cluster of two servers. The

[3]For the sake of readers familiar with Zookeeper and its terminology, in the context of Zookeeper and Zab we use the term "leader" for "primary" and "follower" for "backup" (with no difference in meaning).

[4]Zab contains a preparatory step that optimistically chooses a candidate-leader that already has the up-to-date history, eliminating the need to copy the latest history from one of the backups during startup.

two original members of the cluster hold the latest state, so we want one of them to be elected leader of the new cluster. If one of the three new servers is elected leader, the data stored by the two original members will be lost. (This could happen if the three new servers start up, form a quorum, and elect a leader before the two older servers start up.) Currently, membership changes are done using a "rolling restart" – a procedure whereby servers are shut down and restarted in a particular order so that any quorum of the currently running servers includes at least one server with the latest state. To preserve this invariant, some reconfigurations (in particular, the ones in which quorums from the old and the new configurations do not intersect) require restarting servers multiple times. Service interruptions are unavoidable, as all servers must be restarted at least once. Rolling restart is manually intensive, error prone, and hard to execute correctly even for expert ZooKeeper users (especially if failures happen during reconfiguration). Furthermore, this procedure gives no insight on how clients can discover or react to membership changes.

The protocol we propose in this paper overcomes such problems and enables dynamic changes to the configuration without restarting servers or interrupting the service.

Reconfiguring a state-machine. Primary/backup replication is a special instance of a more general problem, *state-machine replication* (SMR). With SMR, all replicas start from the same state and process the same sequence of operations. Agreement on each operation in the sequence is reached using a consensus protocol such as Paxos [15]. Similarly to our algorithm, most existing SMR approaches use the state-machine itself to change system configuration, that is, the reconfiguration is interjected as any other operation in the sequence of state-machine commands [16]. The details of implementing this in a real system are complex, as pointed out in a keynote describing the implementation of Paxos developed at Google [6]. One of the core difficulties is that a reconfiguration is very different from other SMR commands, in that it changes the consensus algorithm used to agree on the subsequent operations in the sequence.

To better understand the issue, notice that in SMR there is no dependency among operations and thus separate consensus decisions are made for the different "slots" in the history sequence. Thus, if operations 1 through 100 are proposed by some server, it is possible that first operation 1 is committed, then 80, then 20, and so on. It is also possible that an operation proposed by a different server is chosen for slot number 2. Suppose now that a server proposes *reconfiguration* for slot 50. If the proposal achieves a consensus decision, it is most natural to expect that it changes the set of servers that need to execute the consensus algorithm on subsequent slots (51 and onward). Unfortunately, above we stated that we already committed slot number 80 using the current configuration; this could lead to inconsistency (a split brain scenario). We must therefore delay the consensus decisions on a slot until we know the configuration in which it should be executed, i.e., after all previous slots have been decided. As a remedy, Lamport proposed to execute the configuration change α slots in the future, which then allows the consensus algorithms on slots n through $n + \alpha - 1$ to execute simultaneously with slot n. In this manner, we can maintain a 'pipeline' of operations, albeit bounded by α.

Thus, standard SMR reconfiguration approaches limit the concurrent processing of *all* operations, because of the uncertainty introduced by the ability to reconfigure. We use a different approach that overcomes this limitation by exploiting primary order. Our reconfiguration algorithm speculatively executes any number of operations concurrently.

3 Primary/Backup Reconfiguration

We start with a high level description of our reconfiguration protocol. In general, in order for the system to correctly move from a configuration S to a configuration S' we must take the following steps [3], illustrated in Figure 2:

1. persist information about S' on stable storage at a quorum of S (more precisely, a consensus decision must be reached in S regarding the "move" to S');
2. deactivate S, that is, make sure that no further operations can be committed in S;
3. identify and transfer all committed (and potentially committed) state from S to S', persisting it on stable storage at a quorum of S' (a consensus decision in S' regarding its initial state).
4. activate S', so that it can independently operate and process client operations.

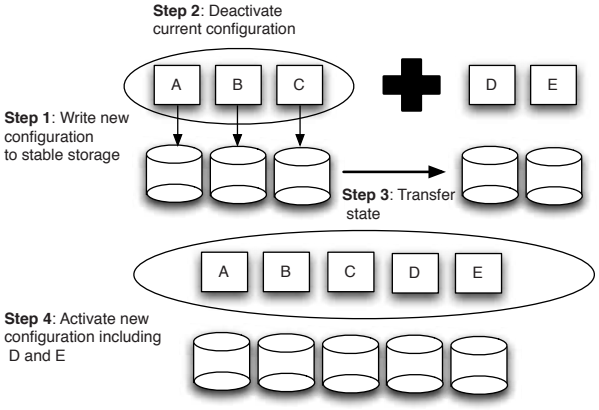


Figure 2: The generic approach to reconfiguration: adding servers D and E to a cluster of three servers A, B and C.

Note that steps 1 and 2 are necessary to avoid split brain. Steps 3 and 4 make sure that no state is lost when moving to S'. The division into four steps is logical and somewhat arbitrary – some of these steps are often executed together.

In a primary/backup system many of the steps above can be simplified by taking advantage of properties already provided by the system. In such systems, the primary is the only one executing operations, producing state-updates which are relative to its current state. Thus, each state-update only makes sense in the context of all previous updates. For this reason, such systems reach agreement on the prefix of updates and not on individual operations. In other words, a new update can be committed only after all previous updates commit. This does not, however, limit concurrency: a primary can execute and send out any number of state-updates speculatively to the backups, however updates are always committed in order and an uncommitted suffix of updates may later be revoked from a backup's log if the primary fails without persisting the update to a sufficient number of replicas (a quorum). Reconfiguration fits this framework well – we interject a configuration update operation, *cop*, in the stream of normal state-updates, which causes a reconfiguration after previously scheduled updates are committed (in state-machine terminology, $\alpha = 1$). Thus, a reconfiguration is persisted to stable storage in the old configuration S just like any other operation in S (this corresponds to step 1 above). At the same time, there is no need to explicitly deactivate S – step 2 follows from the speculative nature of the execution. Just like with any other state-update, the primary may execute any number of subsequent operations, speculatively assuming that *cop* commits. Primary order then makes sure that such operations are committed only after the entire prefix up to the operation (including the configuration change *cop*) is committed, i.e., they can only be committed in the new configuration as required by step 2.

Since the primary is the only one executing operations, its local log includes all state changes that may have been committed; hence, in step 3 there is no need to copy state from other servers. Moreover, we start state transfer ahead of time, to avoid delaying the primary's pipeline. When processing the reconfiguration operation *cop*, the primary only makes sure that state transfer is complete, namely that a quorum of S' has persisted all operations scheduled up to and including *cop*. Finally, in step 4, the primary activates S'.

If the primary of S fails during reconfiguration, a candidate primary in S must discover possible decisions made in step 1. If a new configuration S' is discovered at this stage, the candidate primary must first take steps to commit the stream of commands up to (and including) the operation proposing S', and then it must repeat steps 2–4 in order to transition to S'. Unlike the original primary, the new candidate primary needs to perform a startup-phase in S' and discover the potential actions of a previous primary in S' as well. This presented an interesting challenge in the Zab realm, since a primary in Zab usually has the most up-to-date prefix of commands, and enforces it on the backups. However, a new primary elected from S might have a staler state compared to servers in S'. We must therefore make sure that no committed updates are lost without introducing significant changes to Zab. Below (in Section 3.1), we describe the solution we chose for this pragmatic issue and the *Activation Property* it induces.

We now dive into the details of our protocol. Due to space limitations, we omit the formal proofs here and focus on the intuition behind our algorithm.

3.1 Stable primary

We start by discussing the simpler case, where the primary P of the current configuration S does not fail and continues to lead the next configuration. Figure 3 depicts the flow of the protocol.

pre-step: In order to overlap state-transfer with normal activity, backups in S' connect to the current primary, who initializes their state by transferring its currently committed prefix of updates H. With Zab, such state-transfer happens automatically once backups connect to the primary, and they continue receiving from P all subsequent commands (e.g., op_1 and op_2 in Figure 3), making the transition to S' smooth.

step 1: The primary p schedules *cop*, the reconfiguration command, at the tail of the normal stream of updates. It sends an ACCEPT message containing *cop* to all the backups connected to it (a backup may belong to S and/or to S') and waits for acknowledgments. Consensus on the next configuration is reached once a quorum of S acknowledges *cop*.

step 2: The primary does not stall operations it receives after *cop*. Instead, they are executed immediately and scheduled after *cop*. In principle, all updates following *cop* are the responsibility of S'.

step 3: Transfer of commands has already been initiated in the pre-step; now, p waits for acknowledgement for *cop* and the history of commands which precede it from a quorum of S'.

step 4: Once *cop* is acknowledged by both S and S', the primary commits *cop* and activates S' by sending an ACTIVATE message to backups. Similarly to an ACCEPT, ACTIVATE includes the primary's epoch e and processed by a backup only if e is greater or equal to this backup's $e_{prepare}$.

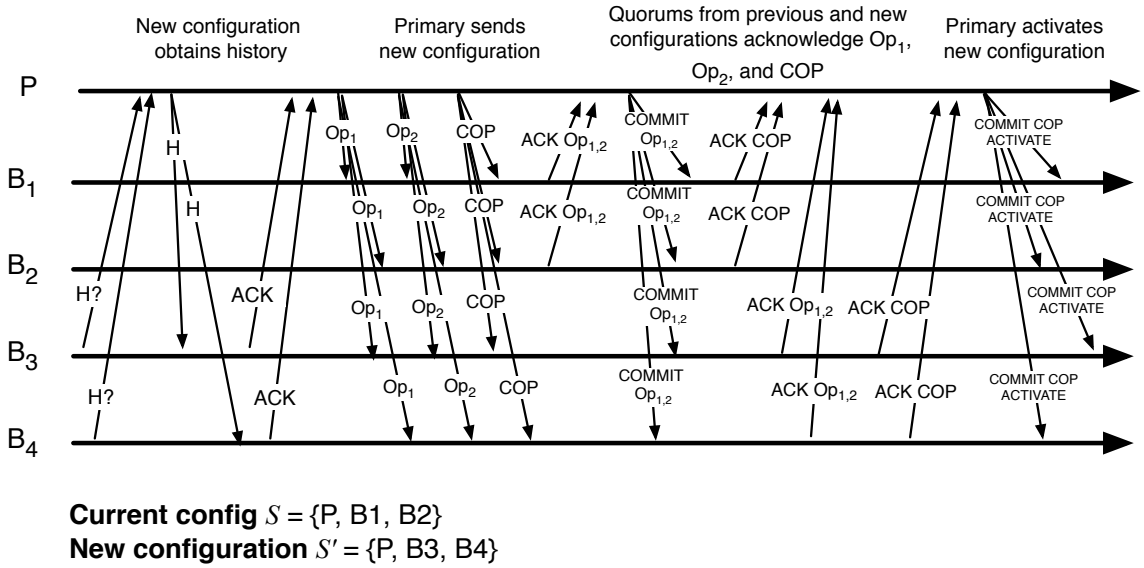


Figure 3: Reconfiguration with a stable primary P.

As mentioned earlier, in order to be compatible with Zookeeper's existing mechanism for recovery from leader failure, we guarantee an additional property:

Activation Property. before ACTIVATE is received by a quorum of S', all updates that may have been committed are persisted to stable storage by a quorum of S.

To guarantee it, we make a change in step 2:

step 2': an update scheduled after *cop* and before the activation message for S' is sent can be committed by a primary in S' only once a quorum of both S and S' acknowledge the update (of course, we also require all preceding updates to be committed). Updates scheduled after the ACTIVATE message for S' is sent, need only be persisted to stable storage by a quorum of S' in order to be committed.

Since the current primary is stable, it becomes the primary of S', and it may skip the startup-phase of a new primary (described in Section 2), since in this case it knows that no updates were committed in S'.

Cascading reconfigurations. Even before ACTIVATE is sent for a configuration S', another reconfiguration operation cop' proposing a configuration S'' may be scheduled by the primary (see Figure 4 below). For example, if we reconfigure to remove a faulty member, and meanwhile detect another failure, we can evict the additional member without ever going through the intermediate step. We streamline cascading reconfigurations by *skipping* the activation of S'.

In the following example, updates u_1 through u_4 are sent by the primary speculatively, before any of them commits, while u_5 is scheduled after all previous updates are committed and the activation message for the last proposed configuration (S'') is sent out.

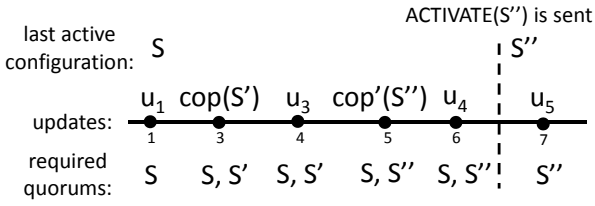


Figure 4: Cascading reconfigurations

Notice that for a given update, only the last active and the last proposed configuration (at the time this update is scheduled) are involved in the protocol steps for that update. Once there is a sufficient window of time between reconfigurations that allows state-transfer to the last proposed configuration to complete, the primary activates that configuration. We note that currently the described extension of the protocol to support multiple concurrent reconfigurations is not being integrated into Zookeeper; for simplicity, a reconfiguration request is rejected if another reconfiguration is currently in progress. (The issuing client may resubmit the reconfiguration request after the current reconfiguration operation completes.)

3.2 Primary failure or replacement

Until now, we assumed that the primary does not fail during the transition to S' and continues as the primary of S'. It remains to ensure that when it is removed or fails, safety is still guaranteed. First, consider the case that

the current primary in S needs to be replaced. There are many reasons why we may want to replace a primary, e.g., the current primary may not be in S', its new role in S' might not allow it to continue leading, or even if the IP address or port it uses for communication with the backups needs to change as part of the reconfiguration.

Our framework easily accommodates this variation: The old primary can still execute operations scheduled after *cop* and send them out to connected backups but it does not commit these operations, as these logically belong in S'. It is the responsibility of a new primary elected in S' to commit these operations. As an optimization, we explicitly include in an ACTIVATE message the identity of a designated, initial primary for S' (this is one of the backups in S', which has acknowledged the longest prefix of operations, including at least *cop*). As before, this primary does not need to execute the startup-phase in S' since we know that no primary previously existed in S'. Obviously, if that default primary fails to form a quorum, we fall-back to the normal primary election in S'.

Likewise, the case of a primary failure after S' has been activated is handled as a normal Zab leader re-election.

An attempted reconfiguration might not even reach a quorum of backups in S, in which case it may disappear from the system like any other failed command.

We are left with the interesting case when a primary-candidate b in S discovers a pending attempt for a consensus on $cop(S')$ by the previous primary. This can mean either that *cop* was already decided, or simply that some backup in the quorum of b heard *cop* from p. As for any other command in the prefix b learns, it must first commit *cop* in S (achieving the consensus decision required in step 1). However, executing *cop* requires additional work, and b must follow the reconfiguration steps to implement it.

The only deviation from the original primary's protocol is that b must follow the startup-phase of a new primary (Section 2) in both S and S'. In order to do so, b connects to the servers in S'. When connecting to a server b' in S', b finds out whether b' knows of the activation of S' (or a later configuration). If S' has been activated, servers in S' may know of newer updates unknown to b, hence b should not attempt to perform state transfer (otherwise it may cause newer updates to be truncated). Instead, b restarts primary re-election in S' (and in particular connects to an already elected primary in S' if such primary exists). Otherwise, b implicitly initiates state-transfer to b' (much like its predecessor did). This includes at least all updates up to *cop* but may also include updates scheduled by the previous primary after *cop*.

This leads us to a subtle issue resulting from our desire to introduce as few changes as possible to the existing implementation of leader recovery in Zookeeper. Recall that the stream of updates by the previous primary may continue past *cop*, and so backups in S' may have a longer history of commands than b. In Zookeeper, connecting to b would cause them to truncate their history. This is exactly why we chose to preserve the *Activation Property*. If b succeeds to connect to a quorum of S' without learning of the activation of S', we know that all updates that may have been committed are stored at a quorum of S. Thus, b will find all such updates once completing the startup-phase in S; in fact, in Zookeeper the candidate b is chosen (by preliminary selection) as the most up-to-date backup in S (that can communicate with a quorum of S), so it will already have the full prefix and no actual transfer of updates is needed during the startup-phase.

Finally, note that b might discover more than a single future reconfiguration while performing its startup-phase in S. For example, it may see that both S' and S'' were proposed. b may in this case skip S' and run the startup-phase in S and S'', after which it activates S''.

3.3 Progress guarantees

As in [2], the fault model represents a dynamic interplay between the execution of reconfiguration operations and the "adversary": The triggering of a reconfiguration event from S to S' marks a first transition. Until this event, a quorum of S is required to remain alive in order for progress to be guaranteed. After it, both a quorum of S and of S' are required to remain alive. The completion of a reconfiguration is generally not known to the participants in the system. In our protocol, it occurs when the following conditions are met: (a) a quorum of S' receives and processes the ACTIVATE message for S', and (b) all operations scheduled before S' is activated by a primary are committed. The former condition indicates that S' can independently process new operations, while the latter indicates that all previous operations, including those scheduled while the reconfiguration was in progress, are committed (it is required due to the *Activation Property* and step 2'). Neither conditions are externally visible to a client or operator submitting the reconfiguration command. However, there is an easy way to make sure that both condition are met: after the reconfiguration completes at the client, it can submit a no-op update operation; once it commits, we know that both conditions (a) and (b) are satisfied (the no-op update can be automatically submitted by the client-side library). An alternative way to achieve this is to introduce another round to the reconfiguration protocol (which, for simplicity and compatibility with Zab, we decided to avoid). Either way, once (a) and (b) are satisfied, the fault model transitions for the second time: only a quorum of S' is required to survive from now on.

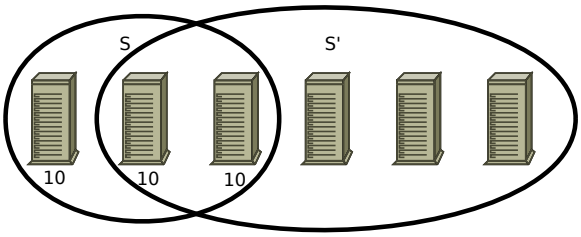


Figure 5: A balanced service (10 clients are connected to each server) about to move to a new configuration S'.

4 Reconfiguring the Clients

Once servers are able to reconfigure themselves we are left with two problems at the client. First, clients need to learn about new servers to be able to connect to them. This is especially important if servers that a client was using have been removed from the configuration or failed. Second, we need to rebalance the load on the servers. ZooKeeper clients use long-lived connections and only change the server they are connected to if it has failed. This means that new servers added to a configuration will not take on new load until new clients start or other servers fail. We can solve the first problem using DNS and by having clients subscribe to configuration changes (see Section 5) in Zookeeper. For lack of space here we concentrate on the second problem.

Figure 5 shows a balanced service with configuration S that is about to move to S'. There are 30 clients in the system and each of the three servers in S serves 10 of the clients. When we change to S' we would like to make sure the new system is also load-balanced. In this example this means that each server should service 6 clients. We would also like to move as few clients as possible since session reestablishment puts load on both the clients and the servers and increases latency for client requests issued while the reestablishment is in process. A final goal is to accomplish the load balance using only logic at the clients so as not to burden the servers.

We denote by M the set of servers that are in both configurations, $S \cap S'$. Machines that are in the old configuration S but not in the new configuration we will label O, that is, $O = S \setminus M$. Machines that are in the new configuration S' but not in the old configuration are labeled N, that is, $N = S' \setminus M$. Denote the total number of clients by C. The number of clients connected to server i in S is denoted by *l(i, S)*.

In general, for a server $i \in S'$, the expected number of clients that connect to i in S', *E(l(i, S'))* is the number of clients connected to it in S plus the number of clients migrating from other servers in S to i (we denote a move from server j to server i by $j \to i$ and a move to any of the servers in a set G by $j \to G$) minus the number of clients migrating from i to other servers in S':

$$E(l(i,S')) = l(i,S) + \sum_{j \in S \wedge j \neq i} l(j,S) * Pr(j \to i) - l(i,S) \sum_{j \in S' \wedge j \neq i} Pr(i \to j)$$

We solve for the probabilities assuming that the load was uniform across all servers in S and requiring that the expected load remains uniform in S' (in the example of Figure 5, we require that $E(l(i,S')) = 6$). Intuitively, the probability of a client switching to a different server depends on whether the cluster size increases or shrinks, and by how much. We have two cases to consider:

Case 1: $|S| < |S'|$ Since the number of servers is increasing, load must move off from all servers. For a server $i \in M$ we get: $E(l(i,S')) = l(i,S) - l(i,S) * Pr(i \to N)$. We can substitute $l(i,S) = C/|S|$ since load was balanced in S, and $E(l(i,S')) = C/|S'|$ since this is what we would like to achieve. This gives:

Rule 1. *If $|S| < |S'|$ and a client is connected to M, then with probability $1 - |S|/|S'|$ the client disconnects from its server and then connects to a random server in N. That is, the choice among the servers in N is made uniformly at random.*

Notice that clients connected to servers in O should move only to N as servers in M have too many clients to begin with.

Rule 2. *If $|S| < |S'|$ and a client is connected to O, then the client moves to a random server in N.*

Case 2: $|S| \geq |S'|$ Since the number of servers decreases or stays the same, the load on each server in S' will be greater or equal to the load on each server in S. Thus, a server in M will not need to decrease load:

Rule 3. *If $|S| \geq |S'|$ and a client is connected to a server in M, it should remain connected.*

The total collective load in S' on all servers in M is the load on M in S plus the expected number of clients that move to M from O:

$$\frac{|M|C}{|S'|} = \frac{|M|C}{|S|} + \frac{|O|C}{|S|} * Pr(i \to M | i \in O)$$

We thus get our last rule:

Rule 4. *If $|S| \geq |S'|$ and a client is connected to a server in O, it moves to a random server in M with probability $\frac{|M|(|S|-|S'|)}{|S'||O|}$; otherwise, moves to a random server in N.*

By having each client independently apply these rules, we achieve uniform load in a distributed fashion.

5 Implementation and Evaluation

We implemented our server and client-side protocols in Apache Zookeeper. To this end we updated the server-side library of ZooKeeper (written in Java) as well as the two client libraries (written in Java and in C). We

added a *reconfig* command to the API that changes the configuration, a *config* command that retrieves the current configuration and additionally allows users to subscribe for configuration changes and finally the *update-server-list* command that triggers the client migration algorithm described in Section 4. We support two reconfiguration modes. The first is incremental – it allows adding and removing servers to the current configuration. The second type of reconfiguration is non-incremental, which means that the user specifies the new configuration. This method allows changing the quorum system dynamically. We allow adding and removing servers as well as changing server roles. We also support dynamically changing the different network addresses and ports used by the system.

In the remainder of this section we evaluate the impact of reconfigurations on Zookeeper clients. We focus on the effect on throughput and latency of normal operations as well as on load balancing.

We performed our evaluation on a cluster of 50 servers. Each server has one Xeon dual-core 2.13GHz processor, 4GB of RAM, gigabit ethernet, and two SATA hard drives. The servers run RHEL 5.3 using the ext3 file system. We use the 1.6 version of Sun's JVM.

We used the Java server configured to log to one dedicated disk and take snapshots on another. Our benchmark client uses the asynchronous Java client API, and each client is allowed up to 100 outstanding requests. Each request consists of a read or write of 1K of data (typical operation size). We focus on read and write operations as the performance of all the operations that modify the state is approximately the same, and the performance of non state modifying operations is approximately the same. When measuring throughput, clients send counts of the number of completed operations every 300ms and we sample every 3s. Finally, note that state-transfer is always performed ahead of time and a reconfig operation simply completes it, thus our measurements do not depend on the size of the Zookeeper database.

Throughput. We first measure the effect of dynamic reconfigurations on throughput of normal operations. To this end, we used 250 simultaneous clients executing on 35 machines, up to 11 of which are dedicated to run Zookeeper servers (typical installations have 3-7 servers, so 11 is larger than a typical setting). Figure 6 shows the throughput in a saturated state as it changes over time. We show measurements for workloads with 100%, 50%, 30% and 15% write operations. The ensemble is initially composed of 7 servers. The following reconfiguration events are marked on the figure: (1) a randomly chosen follower is removed; (2) the follower is added back to the ensemble; (3) the leader is removed; (4) former leader is added back to the ensemble as a follower; (5) a randomly chosen follower is removed, and (6) the follower is added back to the ensemble.

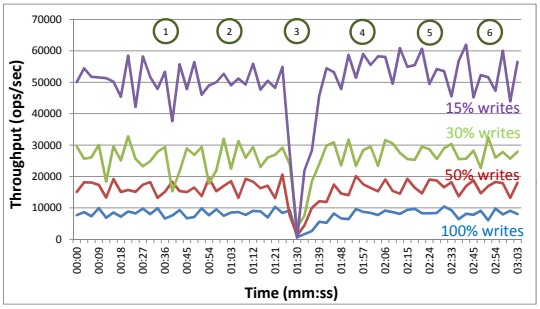


Figure 6: Throughput during configuration changes.

Unsurprisingly, removing the leader has the most significant effect on throughput. In Zookeeper, any leader change (e.g., due to the failure of the previous leader) always renders the system temporarily unavailable, and a reconfiguration removing the leader is no different in that respect. Note that in Zookeeper, each follower is connected only to one leader. Thus, when the leader changes, followers disconnect from the old leader and only after a new leader is established can submit further operations. While this explains why write operations cannot be executed in the transition period (and the throughput drop for a 100% write workload), the reasons for disabling any read activity during leader election (which causes the throughput drop for read intensive workloads) are more subtle. One of the reasons is that Zookeeper guarantees that all operations complete in the order they were invoked. Thus, even asynchronous invocations by the same thread have a well defined order known in advance to the programmer. Keeping this in mind, consider a read operation that follows a write by the same client (not necessarily to the same data item). The read will only be able to complete after the write, whereas writes await the establishment of a new leader[5].

The throughput quickly returns to normal after a leader crash or removal. Notice that read intensive workloads are more sensitive to removal and addition of followers. This is due to the effect of client migration to other followers for load balancing (we explore load-balancing further in Section 5.1). Still, the change in throughput with such reconfigurations is insignificant compared to normal fluctuations of system throughput. The reason is the in-order completion property of Zookeeper mentioned above; writes, which are broadcasted by the leader to followers, determine the throughput of the system. More precisely, the network interface of the leader is the bottleneck. Zookeeper uses a single IP address for leader-follower communication. The throughput of

[5]In Zookeeper 3.4, each operation is blocked until every operation (not necessarily by the same client) previously submitted to the same follower completes; this is not necessary to guarantee the in-order completion semantics and may therefore change in the future.

the system therefore depends on the number of servers connected to the leader, not the number of followers in the ensemble. Note, however, that removing or adding a server from the cluster using the reconfig command does not necessarily change the number of connections. Although a removal excludes a server from participating in Zab voting it does not necessarily disconnect the follower from the leader; an administrator might want to first allow clients to gracefully migrate to other followers and only then disconnect a removed follower or shut it down. In addition, removing a follower is sometimes necessary as an intermediate step when changing its role in the protocol (for example, in some situations when converting an observer to a follower). Figure 7 illustrates this point. It shows two executions, with 30% writes, 250 clients and 11 servers initially in the cluster. There are two reconfiguration events, each removes multiple servers from the cluster. In one execution, the removed servers are turned off while in the other (similarly to Figure 6) removed followers maintain their connections to the leader. The graph shows that disconnecting the servers indeed increases system throughput. This shows, that over-provisioning a cluster by adding more replicas (even if those replicas are observers) can be detrimental to Zookeeper throughput. A better strategy is to reconfigure the system dynamically with changing load.

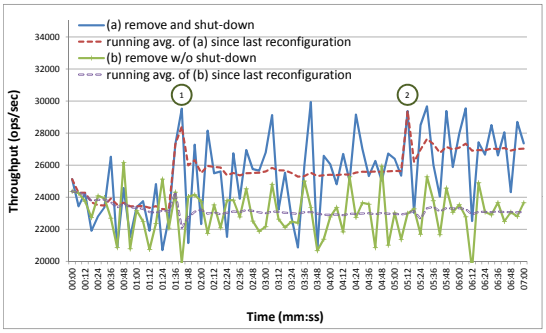


Figure 7: Throughput during configuration changes. Initially there are 11 servers in the cluster. The workload includes 30% writes. Configuration changes: (1) four followers are removed, (2) two additional followers are removed.

Latency. Next, we focus on the effect of reconfiguration on the latency of other requests. We measured the average latency of write operations performed by a single client connected to Zookeeper; the writes are submitted in batches of 100 operations, after all previously submitted writes complete. Initially, the cluster contains seven replicas and writes have an average latency of 10.8ms[6].

We then measured the impact of removing replicas on latency. A client submits a reconfiguration request to remove four randomly chosen followers which is immediately followed by a second write batch. If we use the reconfiguration procedure described in Section 3, we get an average latency again of 10.8ms. However, if we stall the request pipeline during the reconfiguration, the average latency increases to 15.2ms.

With three replicas, our average write latency is 10.5ms. The client then requests to add back four replicas, followed by another write batch. Using our approach write latency is at 11.4ms and jumps to 18.1ms if we stall the pipeline.

Leader removal. Finally, we investigate the effect of reconfigurations removing the leader. Note that a server can never be added to a cluster as leader as we always prioritize the current leader. Figure 8 shows the advantage of designating a new leader when removing the current one, and thus avoiding leader election. It depicts the average time to recover from a leader crash versus the average time to regain system availability following the removal of the leader. The average is taken on 10 executions. We can see that designating a default leader saves up to 1sec, depending on the cluster size. As cluster size increases, leader election takes longer while using a default leader takes constant time regardless of the cluster size. Nevertheless, as the figure shows, cluster size always affects total leader recovery time, as it includes synchronizing state with a quorum of followers.

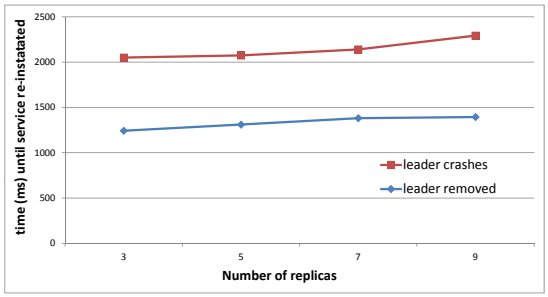


Figure 8: Unavailability following leader removal or crash.

5.1 Load Balancing

In this section, we would like to evaluate our approach for load balancing clients as part of configuration changes. To this end, we experiment with a cluster of nine servers and 1000 clients. Clients subscribe to configuration changes using the *config* command and update their list of servers using the *update-server-list* command when notified of a change. In order to avoid mass migration of clients at the same time, each client waits for a random period of time between 0 and 5sec. The graphs presented below include four reconfiguration events: (1) remove one random server; (2) remove two random servers; (3) remove one random server and add the three previously removed servers, and (4) add the server removed in step 3.

[6]the average latencies presented here are taken over 150 executions or the described experiment and lie within 0.3ms of the real average with 95% confidence

We evaluate load balancing by measuring the minimum and maximum number of clients connected to any of the servers and compare it to the average (number of clients divided by the current number of servers). When the client connections are balanced across the servers, the minimum and maximum are close to the average, i.e., there are no overloaded or under-utilized servers.

Baseline. Our first baseline is the current implementation of load balancing in Zookeeper. The only measure of load is currently the number of clients connected to each server, and Zookeeper is trying to keep the number of connections the same for all servers. To this end, each client creates a random permutation of the list of servers and connects to the first server on its list. If that server fails, it moves on to the next server on the list and so on (in round robin). This approach works reasonably well when system membership is fixed, and can easily accommodate server removals. It does not, however, provide means for incorporating a new server added to the cluster. In order to account for additions in this scheme, we replace the client's list with a new list of servers. The client maintains its connection unless its current server is not in the new list. Figure 9 shows that load is balanced well as long as we perform removals (steps 1 and 2), however when servers are added in steps 3 and 4 the newly added servers are under-utilized. In the beginning of step 3 there are six servers in the system, thus approximately 166 clients are connected to every server. When we remove a server and add three new ones in step 3, the clients connected to the removed server migrate to a random server in the new configuration. Thus, every server out of the eight servers in the new configuration gets an expected 21 additional clients (the newly added servers will only have these clients, as no other clients disconnect from their servers). In step 4 we add back the last server, however no clients migrate to this server. Although all clients find out about the change and update their lists, no client disconnects from its server as it is still part of the system.

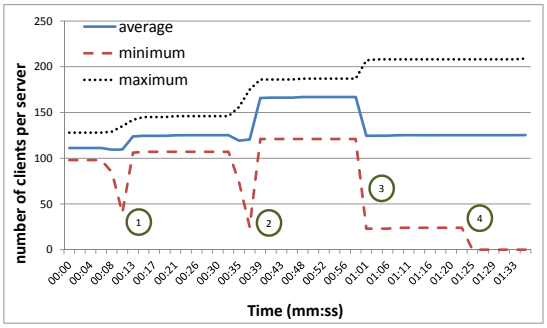


Figure 9: Baseline load balancing.

To mitigate the problem illustrated in Figure 9 we could of course disconnect all clients and re-connect them to randomly chosen servers in the new configuration. This, however, creates excessive migration and unnecessary loss of throughput. Ideally, we would like the number of migrating clients to be proportional to the change in membership. If only a single server is removed (or added), only clients that were (or should be) connected to that server should need to migrate.

Consistent Hashing. A natural way to achieve such limited migration, which we use as a second baseline, is to associate each client with a server using consistent hashing [14]. Client and server identifiers are randomly mapped to points in an m-bit space, which can be seen as circular (i.e., 0 follows $2^m - 1$). Each client is then associated with the server that immediately follows it in the circle. If a server is removed, only the clients that are associated with it will need to migrate by connecting to the next server on the circle. Similarly, if a new server is added a client migrates to it only if the new server was inserted between the client and the server to which it is currently connected. In order to improve load balancing, each server is sometimes hashed k times (usually k is chosen to be in the order of $log(N)$, where N is the number of servers). To evaluated the approach, we implemented it in Zookeeper. Figure 10 shows measurements for $k = 1$, $k = 5$ and $k = 20$. We used MD5 hashing to create random identifiers for clients and servers ($m = 128$). We can see that higher values of k achieve better load balancing. Note, however, that load-balancing in consistent hashing is uniform only with "high probability", which depends on N and k. In the case of Zookeeper, where 3-7 servers (N) are usually used, the values of N and k are not high enough to achieve reasonable load balancing.

Probabilistic Load Balancing. Finally, Figure 11 shows measurements of load-balancing with the approach we have implemented in Zookeeper as outlined in Section 4. Unlike consistent hashing, in this approach *every* client makes a probabilistic decision whether and where to migrate, such that the expected number of clients per server is the same for every server. As we can see from the figure the difference in number of clients between the server with the most clients and the least clients is very small. Using our simple case-based probabilistic load balancing we are able to achieve very close to optimal load-balance using logic entirely at the client.

6 Related Work

Primary order is commonly guaranteed by Primary/Backup replication systems, e.g., Chubby [5], GFS [8], Boxwood [19], PacificA [21], chain replication [20], Harp [17] and Echo [11]. Although Paxos does not guarantee primary order [13], some systems

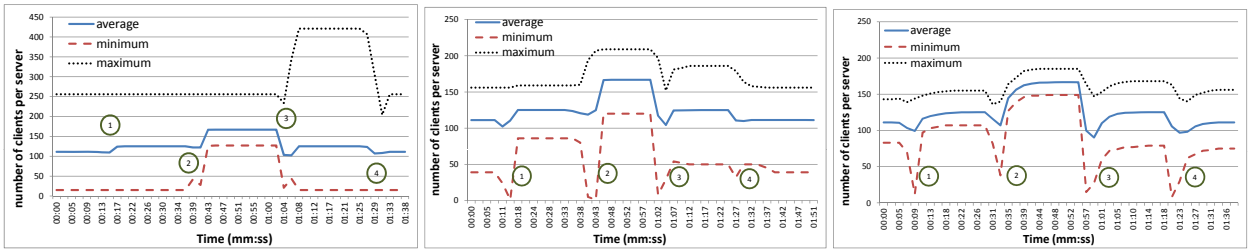


Figure 10: Load balancing using consistent hashing, with $k = 1$ (left), $k = 5$ (middle), and $k = 20$ (right).

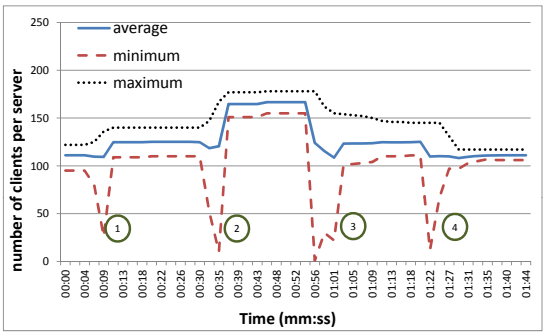


Figure 11: Load balancing using our method (Section 4).

implementing Paxos (such as Chubby and Boxwood) have one outstanding decree at-a-time, which in fact achieves primary-order. This is done primarily to simplify implementation and recovery [19]. Unlike such approaches, we do not limit the the concurrent processing of operations.

Unlike systems such as RAMBO [9], Boxwood [19], GFS [8], Chubby [5], chain replication [20] and PacificA [21] that use an external reconfiguration service, we use the system itself as the reconfiguration engine, exploiting the primary order property to streamline reconfigurations with other operations. Zookeeper is often used by other systems for the exact same purpose, and thus relying on another system for reconfiguring Zookeeper would simply push the problem further as well as introduce additional management overhead. An additional difference from RAMBO is that in our design, every backup has a single "active" configuration in which it operates, unlike in RAMBO where servers maintain a set of possible configurations, and operate in all of them simultaneously. Finally, RAMBO and several other reconfigurable systems (see [1] for a survey), are designed for reconfiguring read/write storage, whereas Zookeeper provides developers with arbitrary functionality, i.e., a universal object via consensus [10]; the read/write reconfiguration problem is conceptually different [2] than the one we address in this paper.

SMART [18] is perhaps the most practical implementation of Paxos [15] SMR published in detail. SMART uses Lamport's α parameter to bound the number of operations that may be executed concurrently (see Section 2). In addition, SMART uses configuration-specific replicas: if the cluster consists of replicas A, B, and C and we are replacing C with D, SMART runs two replicas of A and two of B, one in the new configuration and one in the old, each running its own instance of the replication protocol. An important design consideration in our work has been to introduce minimal changes to Zookeeper, as it is used in production by many commercial companies. Dynamically creating additional Zookeeper replicas just for the purpose of reconfiguration adds an implementation and management overhead that would not be acceptable to Zookeeper users. Unlike SMART, we do not limit concurrency or require any additional resources to reconfigure.

FRAPPE [4] proposes a different solution. Each server in FRAPPE works with a set of possible configurations, similarly to RAMBO. If a reconfiguration is proposed for history slot n, any number of operations can be proposed after n, however their completion is speculative – users are aware that even though these operations commit they may later be rolled back if a different operation is chosen for slot n. This requires servers to maintain a speculative execution tree, each branch corresponding to an assumption on the decision on some reconfiguration for a particular history slot. In case the reconfiguration is chosen for slot n and once state transfer is complete, the speculative operations become permanently committed and the corresponding tree-branch is merged into the "trunk". Otherwise, the branch is simply abandoned. Similarly to SMART and FRAPPE, we do not require any intersection between the memberships of consecutive configurations. The algorithm presented in this paper processes updates speculatively, similar to FRAPPE. However, our algorithm does not require servers to work with or explicitly manage multiple configurations and it does not expose speculative operation completions to the clients.

Group communication systems that provide virtual synchrony [7, 3] are perhaps closer to Zookeeper than Paxos-style replicated state machines. In such systems, a group of processes may exchange messages with others in the group, and the membership of the group (called a view) may change. Virtual synchrony guarantees that all processes transferring from one view to the next agree on the set of messages received in the previous view. Note

that they do not necessarily agree on the order of messages, and processes that did not participate in the previous view do not have to deliver these messages. Still, virtual synchrony is similar to primary order in the sense that it does not allow messages sent in different configurations to interleave just as primary order does not allow messages sent by different leaders to interleave. Unlike state-machine replication systems, which remain available as long as a quorum of the processes are alive, group communication systems must react to every failure by removing the faulty process from the view. While this reconfiguration is in progress, client operations are not processed. Other systems, such as Harp [17] and Echo [11] follow similar methodology, stopping all client operations during reconfigurations. Conversely, our design (similarly to state-machine replication systems) tolerates failures as long as a quorum of the replicas remains available, and allows executing client operations while reconfiguration and state-transfer are in progress.

7 Conclusions

Reconfiguration is hard in general. It becomes especially hard when reconfiguring the configuration service. While intuitively it seems simple, care must be taken to address all failure cases and execution orderings.

Our reconfiguration protocol builds on properties of Primary/Backup systems to achieve high performance reconfigurations without imposing a bound on concurrent processing of operations or stalling them, and without the high management price of previous proposals.

The load balancing algorithm for distributing clients across servers in a new configuration involves decisions made locally at the client in a completely distributed fashion. We guarantee uniform expected load while moving a minimum number of clients between servers.

We implemented our protocols in an existing open-source primary/backup system, and are currently working on integrating it into production. This involved simple changes, mostly to the commit and recovery operations of Zookeeper. Our evaluation shows that there are minimal disruptions in both throughput and latency using our approach.

While the methods described in this paper were implemented in the context of ZooKeeper, the primary order property we have taken advantage of is commonly provided by Primary/Backup systems.

Acknowledgments

We would like to thank Marshall McMullen for his valuable contributions to this project. We thank the Zookeeper open source community and in particular to Vishal Kher, Mahadev Konar, Rakesh Radhakrishnan and Raghu Shastry for their support, helpful discussions, comments and thorough reviews of this work. Finally, we would like to thank the anonymous reviewers and our shepherd, Christopher Small, for their comments.

References

[1] AGUILERA, M. K., KEIDAR, I., MALKHI, D., MARTIN, J.-P., AND SHRAER, A. Reconfiguring replicated atomic storage: A tutorial. *Bulletin of the EATCS 102* (2010), 84–108.

[2] AGUILERA, M. K., KEIDAR, I., MALKHI, D., AND SHRAER, A. Dynamic atomic storage without consensus. *J. ACM 58*, 2 (2011), 7.

[3] BIRMAN, K., MALKHI, D., AND VAN RENESSE, R. Virtually synchronous methodology for dynamic service replication. Tech. Rep. 151, MSR, Nov. 2010.

[4] BORTNIKOV, V., CHOCKLER, G., PERELMAN, D., ROYTMAN, A., SHACHOR, S., AND SHNAYDERMAN, I. Frappé : Fast replication platform for elastic services. In *ACM LADIS* (2011).

[5] BURROWS, M. The chubby lock service for loosely-coupled distributed systems. In *OSDI* (2006), pp. 335–350.

[6] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live: an engineering perspective. In *PODC* (2007), pp. 398–407.

[7] CHOCKLER, G., KEIDAR, I., AND VITENBERG, R. Group communication specifications: a comprehensive study. *ACM Comput. Surv. 33*, 4 (2001), 427–469.

[8] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *SOSP* (2003), pp. 29–43.

[9] GILBERT, S., LYNCH, N. A., AND SHVARTSMAN, A. A. Rambo: a robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing 23*, 4 (2010), 225–272.

[10] HERLIHY, M. Wait-free synchronization. *ACM Trans. Program. Lang. Syst. 13*, 1 (1991), 124–149.

[11] HISGEN, A., BIRRELL, A., JERIAN, C., MANN, T., SCHROEDER, M., AND SWART, G. Granularity and semantic level of replication in the echo distributed file system. In *Proceedings of the IEEE Workshop on the Management of Replicated Data* (November 1990).

[12] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technology Conference (2010)* (2010).

[13] JUNQUEIRA, F. P., REED, B. C., AND SERAFINI, M. Zab: High-performance broadcast for primary-backup systems. In *DSN* (2011), pp. 245–256.

[14] KARGER, D. R., LEHMAN, E., LEIGHTON, F. T., PANIGRAHY, R., LEVINE, M. S., AND LEWIN, D. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC* (1997), pp. 654–663.

[15] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (1998), 133–169.

[16] LAMPORT, L., MALKHI, D., AND ZHOU, L. Reconfiguring a state machine. *SIGACT News 41*, 1 (2010), 63–73.

[17] LISKOV, B., GHEMAWAT, S., GRUBER, R., JOHNSON, P., SHRIRA, L., AND WILLIAMS, M. Replication in the harp file system. In *SOSP* (1991), pp. 226–238.

[18] LORCH, J. R., ADYA, A., BOLOSKY, W. J., CHAIKEN, R., DOUCEUR, J. R., AND HOWELL, J. The smart way to migrate replicated stateful services. In *EuroSys* (2006), pp. 103–115.

[19] MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI* (2004), pp. 105–120.

[20] VAN RENESSE, R., AND SCHNEIDER, F. B. Chain replication for supporting high throughput and availability. In *OSDI* (2004), pp. 91–104.

[21] WEI LIN, MAO YANG, L. Z., AND ZHOU, L. Pacifica: Replication in log-based distributed storage systems. Tech. Rep. MSR-TR-2008-25, MSR, Feb. 2008.

[22] YIN, Z., MA, X., ZHENG, J., ZHOU, Y., BAIRAVASUNDARAM, L. N., AND PASUPATHY, S. An empirical study on configuration errors in commercial and open source systems. In *SOSP* (2011), pp. 159–172.

Surviving congestion in geo-distributed storage systems

Brian Cho
University of Illinois at Urbana-Champaign

Marcos K. Aguilera
Microsoft Research Silicon Valley

Abstract. We present Vivace, a key-value storage system for web applications that span many geographically-distributed sites. Vivace provides strong consistency and replicates data across sites for access locality and disaster tolerance. Vivace is designed to cope well with network congestion across sites, which occurs because the bandwidth across sites is smaller than within sites. To deal with congestion, Vivace relies on two novel algorithms that prioritize a small amount of critical data to avoid delays due to congestion. We evaluate Vivace to show its feasibility and effectiveness.

1 Introduction

Web applications such as web mail, search, online stores, portals, social networks, and others are increasingly deployed in *geo-distributed* data centers, that is, data centers distributed over many geographic locations. These applications often rely on key-value storage systems [24] to keep persistent user data, such as shopping carts, profiles, user preferences, and others. These data should be replicated across data centers or *sites*, to provide disaster tolerance, access locality, and read scalability.

Traditional replication solutions fall in two categories:

- *Local replication,* designed for local-area networks, which provide strong consistency and easy-to-use semantics. Protocols for local replication include ABD and Paxos [16, 37]. These protocols perform poorly when replicas are spread over remote sites.
- *Remote replication,* designed for replicas at many sites, which provides good performance in that case, by propagating updates asynchronously (e.g., [24, 23, 40]). These protocols provide weaker forms of consistency, such as eventual consistency [51].

Vivace is a key-value storage system that combines the advantages of both categories, by providing strong consistency, while performing well when deployed across multiple remote sites. Strong consistency is desirable: although weak consistency may be adequate in some cases (e.g., [24, 23]), strong consistency is necessary in others, when reading stale data is undesirable.[1] The difficulty with providing strong consistency is that it requires coordination across sites to execute storage requests. At first glance, this coordination appears to be prohibitive, due to the higher network latencies across sites. However, typical round-trip latencies across sites are not too high—around 50–200 ms—which can be reasonable for certain storage systems. In fact, Megastore [17] has demonstrated that synchronous replication can sometimes work across sites. But the real problem occurs when the cross-site links become congested, causing the round-trip latency to increase to several seconds or more—as observed in systems like Amazon's EC2 (Section 2). In that case, Megastore and existing synchronous replication solutions suffer from delays of several seconds, which are unacceptable to users. A study by Nielsen [45] indicates that web applications should respond to users within one second or less.

A solution to this problem is to avoid network congestion, by provisioning the cross-site links for peak usage. This solution can be expensive and wasteful, especially when the system operates at peak usage only rarely, as in many computer systems. A better solution is to provision cross-site links more conservatively—say, for typical usage—and design the system to deal with congestion when it appears. This is the approach taken in Vivace.

The key innovation in Vivace is two novel strongly consistent replication algorithms that behave well under congestion. The first algorithm is simpler and implements a storage system with read and write atomic operations. The second algorithm is more complex and implements a state machine [50], which supports generic read-modify-write atomic operations. These algorithms rely on network prioritization of a few latency-critical messages in the system to avoid delays due to congestion. Because the size and number of prioritized messages is small, only a tiny fraction of the total network bandwidth comprises prioritized data. We evaluate Vivace and its algorithms to show that they are feasible and effective at avoiding delays due to congestion, without consuming resources significantly.

2 Setting and goals

We consider a system with multiple data centers or *sites*, where each site consists of many machines connected by a local-area network with low latency and high bandwidth. Sites are connected to each other via wide-area network links which have lower bandwidth and higher latencies than the links within a site. Machines or processes are subject to crash failures; we do not consider Byzantine failures. Network partitions may prevent communication across sites. Such partitions are rare, be-

[1] For this reason, cloud services such as Google AppEngine, Amazon SimpleDB, and others support both weak and strong consistency.

Mbps	K$/month	Mbps	K$/month	Mbps	K$/month
2	3.3	8	11.6	40	31.2
4	6.4	10	13.7	50	37.3
5	7.8	20	21.4	60	43.4
6	9.1	30	25.1	100	67.9

Figure 1: Sample cost to connect sites using MPLS VPNs, as a function of the contracted maximum bandwidth [5].

cause the cross-site links are either provided by ISPs that promise very high availability, or by private leased lines that also provide very high availability. Nevertheless, partitions may occur; we represent them as larger network delays that last until the partition is healed. The system is subject to disasters that may destroy an entire site or disconnect the site from the other sites. We represent such a disaster as a crash of many or all the machines in the affected site.

Processes have access to synchronized clocks, which are used as counters. These clocks can be realized with GPS sensors, radio signals, or protocols such as NTP. Alternatively, it is possible to replace the use of synchronized clocks in Vivace with a distributed protocol that provides a global counter. However, doing so would reduce performance of Vivace due to the need of a network round-trip to obtain a counter value.

The network is subject to intermittent congestion across sites, because the cross-site bandwidth is provisioned based on typical or average usage, not peak usage. This is so due to cost considerations. For example, small or medium data centers may use MPLS VPNs[2], which can provide a fixed contracted bandwidth priced accordingly (Figure 1). Large data centers might use private leased lines or other solutions, but the bandwidth could still be much smaller than needed at peak times, causing congestion. For example, in Amazon EC2, simple measurements of network latencies across locations show that congestion occurs often, causing the round-trip latencies of messages sent over TCP to grow from hundreds of milliseconds to several seconds or more [36].

We assume the ability to prioritize messages in the network, so that they are transmitted ahead of other messages. We evaluate whether this assumption holds in Section 6.2. Support for prioritization is required only at the network edge of a site—say, at the router that connects the site to the external network—not in the external network itself, because congestion tends to occur at the site edge.

We wish a distributed key-value storage system that replicates data across sites and provides strong consistency, defined precisely by linearizability [33]. Roughly speaking, linearizability requires each storage operation—such as a read or write—to appear to take effect sequentially at an instantaneous point in time.

The key-value storage system should support two types of data objects: (1) RW objects, which provide read-write storage, and (2) RMW objects, which provide state machines; RMW stands for read-modify-write, which are operations that can modify the object's state based on its previous state. Objects of either type are identified by a *key*. A RW object has two operations, *write*(*value*) and *read*(), which stores a new value and retrieves the current value of the object, respectively. The size of object keys tend to be small (bytes), whereas the values stored in an object can be much larger (KBs).

RMW objects have an operation *execute*(*command*), which applies the command. RMW objects are state machines [50]—which are implemented by protocols such as Paxos—and the command can be an arbitrary deterministic procedure that, based on the current state of the object, modifies the state of the object and returns a result. We consider a slightly weaker type of state machine, where if many concurrent commands are executed on the same RMW object, the system is allowed to abort the execution of the commands, returning a special value $\perp$. Aborted operations may or may not take effect; if an aborted operation does not take effect, the user can reissue the operation after a while. By using known techniques, such as exponential random back-off or a leader election service, it is possible to guarantee that the operation takes effect exactly once [11].

3 Design and algorithms

We now explain the design of Vivace, with a focus on the replication algorithms that it uses.

3.1 Architecture

Vivace has a standard architecture for a key-value storage system, which we now briefly describe. There is a set of storage servers, which store the state of objects, and a set of client machines, which run applications. Applications interact with Vivace via a client library.

Each object has a *type*, which is RW or RMW (Section 2), and a *replica set*, which indicates the storage servers and sites where the object is replicated. In addition, for each site, each object has a set of local storage servers, called the *local replica set* (or simply *local set*) of the object, which we explain in Section 3.2. Our algorithms can make progress despite the crash of any minority of replicas in the replica set, plus any minority of replicas in the local set. Replica sets and local sets can be provisioned accordingly. For example, in our evaluation we provisioned three replicas for every replica set and local set, so the system tolerates one failure in each set.

[2]MPLS VPN is a technology offered by ISPs to connect together sites [1].

The type, replica set, and local set comprise the *metadata* of an object. To reduce the overhead of storing metadata, objects are grouped into containers, and all objects in the container share the same metadata. The container of an object is fixed and the container id is a part of the object's key. A directory service stores the metadata for each container. Clients consult the directory service rarely, since they cache the metadata. The directory service is itself implemented using Vivace's replication algorithms, except the metadata for the directory objects is fixed: the type is a RW object, and the replica set is a fixed set of DNS names.

3.2 Algorithm for RW objects

Vivace's algorithm for RW objects is based on the *ABD* algorithm by Attiya, Bar-Noy, and Dolev [16]. It is a simple algorithm that provides linearizable read and write operations that always succeed when a majority of replicas are up. Moreover, the algorithm ensures safety and progress in a completely asynchronous system. In Section 3.3, we present a more complex algorithm that implements a state machine for RMW objects. That algorithm requires some partial synchrony to ensure progress—as with any other state machine algorithm.

We now briefly describe the ABD algorithm. To write value v to an object, the client obtains a new timestamp ts, asks for the replicas to store (v,ts), and waits for a majority of acknowledgments[3]. To read the latest value, the client asks the replicas to send their current pairs of (v,ts). The client waits for a majority of replies, and picks the reply (v',mts) with the largest timestamp mts. The client then executes a write-back phase, in which it asks replicas to store (v',mts) and waits for a majority of acknowledgments. This write-back phase is needed to provide linearizability: it ensures that a subsequent read operation sees v' or a more recent value.

If the replicas are in different sites, the ABD algorithm sends and receives messages across sites before the operation can complete. If remote network paths are congested, this remote communication can take a long time. We propose to avoid these congestion delays, by having the client use prioritized messages that are transmitted ahead of other messages, thereby bypassing the congestion. Prioritized messages must be small, otherwise these messages themselves will congest the network. To obtain small messages, we modify the ABD algorithm by breaking up its messages into two parts: critical fields—such as timestamps, statuses, and acks—that must be sent immediately, and the other fields. We restructure the ABD algorithm to operate correctly when the messages are broken up, and then we use prioritized messages for sending the critical fields. The challenge in doing so is threefold. First, we must still continue to provide linearizability (strong consistency) when the messages have been split; in fact, we define linearizability as the correctness criteria for the new algorithm. The difficulty here is that a message that is split in two parts may be interleaved with the split messages of other clients, creating concurrency problems. To address such problems, the new algorithm includes some additional phases of communication and coordination. Second, we must find a very small amount of critical information to prioritize, otherwise the prioritized data will congest the network; we later analyze and evaluate the new algorithm to show that the prioritized fields we chose indeed comprise a small proportion of the total data sent. Third, we must not impose significant extra overhead in the new algorithm, otherwise it will perform worse than the original algorithm when the network is not congested; in particular, the new algorithm has extra phases of communication; we later evaluate this overhead and show that it is very small and worth the benefit, because the extra communication occurs in the local area network.

We now describe the algorithm in more detail. To read a value, the client uses a small prioritized message to ask replicas to send their current timestamp. Replicas reply with another small prioritized message. Once the client has a majority of replies, it computes the highest timestamp mts that it received. It then asks replicas to send the data associated with timestamp mts. The reply is a large non-prioritized message with data but, in the common case, a replica in the local site has the data, so this replica responds quickly without remote communication. Thus, the client can read the value without being affected by the congestion on the remote path. The client then performs a fast write-back phase, by sending a small prioritized message with only the highest timestamp, not the data value.

To write a value v, the client obtains a new timestamp ts. We want to avoid sending v to remote sites in the critical path. The client stores (v,ts) at temporary replicas located in the same site as the client; the set of temporary replicas is called the *local replica set*, and each replica is called a *local replica*. The client also stores the timestamp ts at the (normal) replicas—which are typically at remote sites—using small prioritized messages; these messages carry only the timestamp ts, not the data value, and they bypass congestion on the remote paths. Once the client receives enough acknowledgments, the operation completes. Meanwhile, in the background, each local replica propagates the data value to the (normal) replicas. These larger messages do not delay the client, even if there is congestion, because they are not in the critical path. Furthermore, the larger messages are not prioritized.

[3] In the algorithm in [16], there are no synchronized clocks, so there is an extra round of communication to obtain a new timestamp. Here we obtain the timestamp from the synchronized clocks.

Algorithm 1 Vivace algorithm for RW objects

```
function read(key):
  acks ← sendwait(∗⟨R-TS, key⟩, nodes[key], ⌈(n+1)/2⌉)
  mts ← max_{1≤i≤⌈(n+1)/2⌉} acks[i].msg.ts
  data ← sendwait(∗⟨R-DATA, key, mts⟩, nodes[key], 1)
  sendwait(∗⟨W-TS, key, data[0].ts⟩, nodes[key], ⌈(n+1)/2⌉)
  return data[0].val

function write(key, val):
  ts ← clock()
  sendwait(⟨W-LOCAL, key, ts, val⟩, local_nodes[key], ⌈(n+1)/2⌉)
  sendwait(∗⟨W-TS, key, ts⟩, nodes[key], ⌈(n+1)/2⌉)

function sendwait(msg, nodes, num_acks):
  send msg to nodes
  wait for num_acks replies
  return the replies in an array acks[0..num_acks − 1]

upon receive ⟨R-TS, key⟩:
  return ∗⟨ACK-R-TS, ts[key]⟩

upon receive ⟨R-DATA, key, ts⟩:
  wait until ts[key] ≥ ts and val[key] ≠ ⊥
  return ⟨ACK-R-DATA, ts[key], val[key]⟩

upon receive ⟨W-TS, key, ts⟩:
  if ts > ts[key] then
    ts[key] ← ts
    if remote_buf[key][ts] exists then
      val[key] ← remote_buf[key][ts]
      delete remote_buf[key][x] for all x ≤ ts
    else val[key] ← ⊥
  return ∗⟨ACK-W-TS⟩

upon receive ⟨W-LOCAL, key, ts, val⟩:
  local_buf[key][ts] ← val
  async
    sendwait(⟨W-REMOTE, key, ts, val⟩, nodes[key], ⌈(n+1)/2⌉)
    delete local_buf[key][ts]
  return ∗⟨ACK-W-LOCAL⟩

upon receive ⟨W-REMOTE, key, ts, val⟩:
  if ts = ts[key] then val[key] ← val
  else if ts > ts[key] then remote_buf[key][ts] ← val
  return ⟨ACK-W-REMOTE⟩
```

Detailed pseudocode. The pseudocode is given by Algorithm 1. We denote by $read(key)$ and $write(key, v)$ the operations to read and write an object with the given *key*; n is the number of replicas for the object, and f is a fault-tolerance parameter indicating the maximum number of replicas that may crash. The algorithm requires that $f < n/2$, that is, only a minority of replicas may crash. The replica set of an object with a given *key* is denoted $nodes[key]$, while the local replica set at a given *site* is denoted $local\_nodes[key, site]$. We omit *site* from $local\_nodes[key, site]$ when the site is local (where the client is). That is, $local\_nodes[key]$ refers to the local replica set at the client's site. A prioritized message m is denoted $*\langle m \rangle$, and a normal message m is denoted $\langle m \rangle$.

The communication in the algorithm occurs via a simple function *sendwait*, which sends a given message *msg* to a set of *nodes* and waits for *num_acks* replies. The replies are returned in an array.

To write a value, the client obtains a new timestamp and executes two phases. In the first phase, the client sends the value and timestamp to the local replicas, and waits for an acknowledgment from a majority. In the second phase, the client sends just the timestamp to the replicas, using prioritized messages. When the client receives acknowledgments from a majority, it completes the write operation. Meanwhile, the local replicas propagate the value to the (regular) replicas. The client need not wait for the propagation to complete, because a majority of the local replicas already store the data and a majority of the replicas store the timestamp: if another client in a different site were to execute a read operation, it would observe the timestamp from at least one replica and know what value it needs to wait for.

To read a value, the client executes three phases. In the first phase, the client retrieves the timestamp from a majority of the replicas, and picks the highest timestamp *mts*. In the second phase, the client asks the replicas to send the data associated with this timestamp, if they have it. A replica replies only if it has a timestamp at least as large as the requested timestamp. This phase completes when the client obtains its first reply. In the common case, this reply arrives quickly from a replica in the same site as the client. Once the second phase has completed, the client knows the value that it must return for the read operation. In the third phase, the client writes back the timestamp to the replicas using prioritized messages. When the client receives a majority of acknowledgments, it completes the read operation.

Note that the client returns from a read operation without having to write back the value v. This is possible because the client that originally sent timestamp *mts* to the replicas did so only after it had stored v at a majority of local replicas. When the read operation returns, a majority of replicas has seen the timestamp *mts* of v, but not necessarily v itself. However, we are guaranteed that a majority of replicas subsequently receive v from the local replicas.

3.3 Algorithm for RMW objects

We now present Vivace's algorithm for state machines, to implement RMW objects. We apply the same principles as in Section 3.2: the basic idea is to break-up the protocol messages into critical and non-critical fields, restructuring the algorithm so that, in the critical path, remote communication involves only the critical fields. The non-critical fields are stored at a majority of local replicas and later propagated to the (regular) replicas in the background.

Our starting point is an algorithm for RMW objects similar to the algorithms in [27, 21, 11], which we now briefly describe—we later explain how we apply the above principles to this algorithm. The base RMW al-

Algorithm 2 Algorithm for RMW objects in a LAN

```
function execute(key, command):
  ots ← clock()
  acks ← sendwait(⟨OR, key, ots⟩, nodes[key], ⌈(n+1)/2⌉)
  if acks = ⊥ then return ⊥
  mts ← max_{1≤i≤⌈(n+1)/2⌉} acks[i].msg.ts
  mval ← acks[i].msg.val where acks[i].msg.ts = mts
  ⟨val, r⟩ ← apply(mval, command)
  w_acks ← sendwait(⟨OW, key, ots, val⟩, nodes[key], ⌈(n+1)/2⌉)
  if w_acks = ⊥ then return ⊥
  else return r

function sendwait(msg, nodes, num_acks):
  send msg to nodes
  wait for num_acks replies
  if any reply has status = false then return ⊥
  else return the replies in an array acks[0..num_acks − 1]

upon receive ⟨OR, key, ots⟩:
  if ots > ots[key] then
    ots[key] ← ots
    return ⟨ACK-OR, true, ts[key], val[key]⟩
  else return ⟨ACK-OR, false⟩
upon receive ⟨OW, key, ots, val⟩:
  if ots ≥ ots[key] then
    ots[key] ← ots
    ts[key] ← ots
    val[key] ← val
    return ⟨ACK-OW, true⟩
  else return ⟨ACK-OW, false⟩
```

Algorithm 3 Vivace algorithm for RMW objects

```
function execute(key, command):
  ots ← clock()
  acks ← sendwait(∗⟨OR-TS, key, ots⟩, nodes[key], ⌈(n+1)/2⌉)
  if acks = ⊥ then return ⊥
  mts ← max_{1≤i≤⌈(n+1)/2⌉} acks[i].msg.ts
  data ← sendwait(∗⟨OR-DATA, key, mts⟩, nodes, 1)
  if data = ⊥ then return ⊥
  ⟨val, r⟩ ← apply(data[0].val, command)
  sendwait(⟨W-LOCAL, key, ots, val⟩, local_nodes[key], ⌈(n+1)/2⌉)
  w_acks ← sendwait(∗⟨OW-TS, key, ots⟩, nodes[key], ⌈(n+1)/2⌉)
  if w_acks = ⊥ then
    return ⊥
  else
    return r

function sendwait(msg, nodes, num_acks):
  send msg to nodes
  wait for num_acks replies
  if any reply has status = false then return ⊥
  else return the replies in an array acks[0..num_acks − 1]

upon receive ⟨OR-TS, key, ots⟩:
  if ots > ots[key] then
    ots[key] ← ots
    return ∗⟨ACK-OR-TS, true, ts[key]⟩
  else
    return ∗⟨ACK-OR-TS, false⟩
upon receive ⟨OR-DATA, key, ts⟩:
  wait until ts[key] ≥ ts and val[key] ≠ ⊥
  if ts[key] = ts then
    return ⟨ACK-OR-DATA, true, val[key]⟩
  else
    return ⟨ACK-OR-DATA, false⟩
upon receive ⟨OW-TS, key, ots⟩:
  if ots ≥ ots[key] then
    ots[key] ← ots
    ts[key] ← ots
    if remote_buf[key][ots] exists then
      val[key] ← remote_buf[key][ots]
      delete remote_buf[key][x] for all x ≤ ots
    else
      val[key] ← ⊥
    return ∗⟨ACK-OW-TS, true⟩
  else
    return ∗⟨ACK-OW-TS, false⟩
upon receive ⟨W-LOCAL, key, ts, val⟩:
  local_buf[key][ts] ← val
  async
    sendwait(⟨W-REMOTE, key, ts, val⟩, nodes[key], ⌈(n+1)/2⌉)
    delete local_buf[key][ts]
  return ∗⟨ACK-W-LOCAL⟩

upon receive ⟨W-REMOTE, key, ts, val⟩:
  if ts = ts[key] then
    val[key] ← val
  else if ts > ts[key] then
    remote_buf[key][ts] ← val
  return ⟨ACK-W-REMOTE⟩
```

gorithm is not new; we explain it here for completeness.

To execute a command, a client first obtains a new timestamp *ts*, and sends it to the replicas. Each replica stores the timestamp as a tentative ordering timestamp and subsequently rejects smaller timestamps. The replica replies with its current value and associated timestamp. The client waits for a majority of responses, and picks the value v with the highest timestamp. It applies the command to the value v to obtain a new value v' and a response r. The client then asks the replicas to store v' with the new timestamp *ts*. Each replica accepts the request if it has never seen a higher timestamp; otherwise, the replica returns an error to the client. The client waits for a majority of responses, and if any of them is an error, the client aborts by returning $\perp$; otherwise, if no responses were an error, the client returns r.

The pseudocode is given by Algorithm 2. The first message sent by a client has a tag *OR*, which asks each replica to store a tentative ordering timestamp and reply with its current value and timestamp. The second message sent by a client has a tag *OW*, which asks each replica to store the new value and timestamp. These messages can have data values that are large, and they are sent in the critical path.

We now explain how to send just small messages in the critical path, so that we can prioritize these messages and make the algorithm go faster when there is congestion. The *OR* message in Algorithm 2 itself does not have a data value, but its response carries data. We modify the algorithm so that the response no longer carries any data, just a timestamp. The client then picks the largest received timestamp *mts* and must now retrieve the value v associated with it, so that it can apply the command to v. To do so, the client sends a separate *OR-DATA*

message to all replicas asking specifically to retrieve the value associated with *mts*. In the common case, a replica at the local site has the appropriate value and replies to the client quickly. The client can now proceed as before, by applying the command to obtain a new value v' and a response r that it will return to the caller when it is finished.

The *OW* message in Algorithm 2 carries the new value v', so we must change it. The client uses the local replica set as in the write function of Algorithm 1. The client proceeds in two phases: it first sends v' to the local replicas in a *W-LOCAL* message and waits for a majority of replies. In the background, the local replicas send the data to the (regular) replicas. The client then sends just the new timestamp *ts* to the replicas, knowing that they will eventually receive the value from the local replicas. Algorithm 3 presents the pseudocode with these ideas.

3.4 Optimizations

We now describe some optimizations to Algorithms 1 and 3, to further reduce the latency and bandwidth of operations.

3.4.1 Read optimizations

We present two optimizations for the three-phase read operation of Algorithm 1. The first optimization reduces the number of phases, while the second removes the need for a client to communicate with a majority of replicas.

Avoiding or parallelizing the write-back phase. Recall that Algorithm 1 has a write-back phase, which propagates the largest timestamp *mts* to a majority of replicas. This phase can be avoided in some common cases. In the original ABD algorithm, if (C1) the client receives the largest timestamp *mts* from a majority of replicas in the first phase, it can skip the write-back phase. Similarly, we can skip this phase in Algorithm 1, provided that the same condition (C1) is met and another condition is also met: (C2) the timestamp of the data received in phase two is also *mts*. (Condition (C2) is not needed in the original ABD algorithm because the data and timestamps are together.) With this optimization, in the common case when there are no failures or concurrent writes, a read completes in two phases. It is also possible to read in two phases when (C1) is not met, but (C2) is. To do this, we add a parallel write-back in the second phase, triggered when the client observes that (C1) does not hold. The client proactively writes back *mts* in phase two, in parallel with the request for data. When a data reply is received, the client checks (C2)—it compares the data timestamp with *mts*—and if it holds the read operation can complete in two phases. There are two rare corner cases, when (C2) does not hold. In that case, if (C1) was met in phase one, the read remains as in Algorithm 1; if neither (C1) nor (C2) hold, then the read can use the parallel write-back in phase two.

Reading data from fewer replicas. This optimization reduces bandwidth usage, by having the client send *R-DATA* requests to only some replicas. After the first read phase, the client considers the set of replicas for which it received *ACK-R-TS* containing the large timestamp *mts*. If a local replica exists in this set, the client sends a single *R-DATA* request to that replica. If not, the client sends the request to a subset of replicas, based on some policy. For example, the client can keep a history of recent latencies at remote links, and send a single request to the replica with the lowest latency. The policy used affects performance, but policy choice is orthogonal to the algorithm.

3.4.2 Role change optimizations

In these optimizations, the role played at a node is moved to another node: the first optimization moves the execution of commands from the client to a replica, and the second moves the client role from outside to inside a replica.

Executing commands at replicas. In the execute operation of Algorithm 3, the client reads the current value of the object from the replicas, applies the command to obtain the new value, and writes that value to the replicas. Doing so involves transferring the value from the replicas to the client and back to the replicas. If the value is large, but the command is small, it is more efficient for the client to send the command to the replicas and thereby avoid transferring the value back and forth. To do this, the client first sends the *OR-TS* message and finds the largest timestamp *mts*—this determines the state on which the command should be applied. Then, rather than retrieving the data, the client sends the command to the replicas and the timestamp *mts*. The replicas apply the command to the value with timestamp *mts* (they may have to wait until they obtain that value, but they eventually do), or they reject the command if they see a larger timestamp. If a replica applies the command, it stores the new value in a temporary buffer together with the new timestamp. The client waits for a majority of responses, and if none of these are rejects, the client can send *OW-TS* messages as before. A replica then retrieves the value from its temporary buffer. This optimization reduces the bandwidth consumption of remote links when the command is smaller than the data value.

Delegating to a replica. In Vivace, the client library does not directly execute Algorithms 1 and 3. Rather, the library contacts one of the replicas, which then executes the algorithms on behalf of the client. Doing so is a common technique that saves bandwidth of the client, at the expense of the added latency of a local round-trip. It also makes it possible to modify the algorithms without changing the client library.

Message type	normal max delay	priority max delay
Local, within site	δ	δ
Remote, across sites	D	d

Figure 2: One-way delay parameters for latency analysis.

4 Analysis

We now analyze the algorithms of Section 3.

4.1 Latency

We first consider latency, when a majority of the replicas are on sites different from the client's. (If a majority of the replicas are in the client's site, clients complete their operation locally.) One-way message delays are represented by a few parameters, depending on whether the delay is within or across sites, and whether the message is normal or prioritized, as shown in Figure 2. Within a site, normal and prioritized messages have the same delay δ, due to lack of congestion. The delay for messages sent to remote sites are represented as D when sent normally, and d when prioritized. All the delay parameters incorporate the time to send, transmit, receive, and process a message.

Figure 3 summarizes the results. An execution of an operation can have different latencies, depending on the set of live replicas and the object state at those replicas. We analyze two cases: common and worst. The common case represents a situation with no failures, so that there is a live replica at the local site (the site where the client is). The worst case occurs when (a) all replicas at the local site are failed, and (b) the latest value is not yet stored at any replicas in the replica set; it is stored only at temporary, local set replicas, in a site remote to the client.

We first consider writes of RW objects. The ABD algorithm has a round-trip between the client and replicas (latency: $2D$). In the new Algorithm 1, this round-trip is replaced by two phases: the first one has a round-trip with local replicas (2δ), and the second one has a prioritized round-trip to remote sites ($2d$). By adding these delays, we obtain the total write latencies in Figure 3.

The read operation of ABD has two phases, each with a round-trip ($2D+2D$). Algorithm 1 has three phases. The first phase has a prioritized round-trip ($2d$). The second phase depends on whether there is a replica at the client's site. If there is, the phase has a local round-trip (2δ); otherwise, there are three message delays: (1) the client sends a prioritized request to the replicas (d); (2) the replicas may not have the most recent value and must wait for it from a remote temporary replica (D); and (3) a remote replica sends the value to the client (D). The final write-back phase has a prioritized round-trip to remote replicas ($2d$). By adding these delays, we obtain the total of read latencies in Figure 3.

Algorithm	Operation	Common Case	Worst Case
ABD Algorithm (prior work)	read write	$4D$ $2D$	$4D$ $2D$
Algorithm 1 (new)	read write	$2\delta+4d$ $2\delta+2d$	$5d+2D$ $2\delta+2d$
Algorithm 2 (prior work)	execute	$4D$	$4D$
Algorithm 3 (new)	execute	$4\delta+4d$	$2\delta+5d+2D$

Figure 3: Message delays: common and worst cases.

From Figure 3, we can see that reads and writes of RW objects are faster with the new Algorithm 1 than with ABD, because typically $\delta \ll d \ll D$. Also note that, in the common case, the latency of Algorithm 1 is independent of D, which is not true for ABD.

We now consider the execute operation for RMW objects. Algorithm 2 has two remote round-trips ($2D+2D$). Algorithm 3 has four phases. The first and fourth phases each have a prioritized round-trip ($2d+2d$). Without failures, the second and third phases have a local round-trip ($2\delta+2\delta$). When replicas at the local site are not live, the read phase is prolonged, as in the read operation of Algorithm 1, which we analyzed above (it takes $d+2D$ instead of 2δ). By adding these delays, we obtain the total of execute latencies in Figure 3. We can see that the new Algorithm 3 is significantly better than Algorithm 2 and that, in the common case, the latency of Algorithm 3 does not depend on D.

4.2 Size of prioritized and normal messages

Prioritized messages should be a small fraction of the traffic, otherwise they become useless. We now analyze whether that is the case with the new algorithms. Prioritized messages in Algorithms 1 and 3 have up to four pieces of information: message type, key, timestamp, and an accept bit indicating if the request is accepted or rejected. The message type and accept bit are stored in one byte. The timestamp is eight bytes, which is large enough so that it does not wrap around. The key has variable length, but is typically small, say 16 bytes. Adding up, the size of a prioritized message is up to 25 bytes. Each data message (which is not prioritized) has the preceding information and a value with several KBs, which is orders of magnitude larger than a prioritized message.

This difference in size is advantageous. Let k be the factor by which normal messages are larger than prioritized messages, and B be the bandwidth of a remote link. Then clients can issue storage operations with peaks of throughput equal to $P = k \times B$ without affecting the performance of the system: at the peak throughput, the entire remote link bandwidth is consumed by prioritized messages, and their message delays are still d, so Algorithms 1 and 3 perform as expected. Since k can be large (with data values of size 1 KB, $k \approx 40$), the benefit is significant.

4.3 Fault tolerance

The new Algorithms 1 and 3 are fault tolerant: they tolerate up to f replica crashes. Even if a site disaster destroys more than f replicas in a site, the algorithms safeguard most of the data: only data written in a small window of vulnerability is lost (data held by the temporary local replicas but not yet propagated remotely). Furthermore, the algorithms allow administrators to identify the lost data quickly: these are the keys for which the remote replicas store a timestamp but not the data itself.

5 Implementation

Vivace consists of 6000 lines of Java. Clients and servers communicate using TCP. The system can be configured to use any of the four algorithms of Section 3: the ABD algorithm, Algorithm 1, Algorithm 2, and Algorithm 3. We implemented the optimizations, in Section 3.4 of avoiding the write-back phase and delegation to a replica, for the experiments in Section 6.4. We did not implement the directory service: currently, the metadata for containers is kept in a static configuration file. This does not affect our performance evaluation, because metadata lookups are rare due to caching.

6 Evaluation

We now evaluate Vivace. After describing the experimental setup (Section 6.1), we validate the assumption we made in Vivace that prioritized messages are feasible and effective (Section 6.2). We then consider the performance of the new algorithms of Vivace (Section 6.3). Finally, we demonstrate the benefits of Vivace in a real web application (Section 6.4).

6.1 Experimental setup

The experimental setup consists of machines in Amazon's EC2 and a private local cluster in Urbana-Champaign. In EC2, we use extra large virtual machine instances with 4 CPU cores and 15 GB of memory, in two locations, Virginia and Ireland. We use the private cluster for experiments that require changing the configuration of the network. The local cluster has three PCs with 2.4 GHz Pentium 4 CPU, 1 GB of memory, and an Intel PRO/100 NIC. The cluster is connected to the Internet via a Cisco Catalyst 3550 router, which has 48 100 Mbps ports. The median round-trip latencies were the following (in ms):

	Local cluster	EC2 Virginia	EC2 Ireland
Local cluster	<1	23	109
EC2 Virginia		<1	93
EC2 Ireland			<1

6.2 Message prioritization schemes

Vivace assumes the existence of an effective mechanism to prioritize messages in the network. In this section, we examine whether this assumption holds. We consider network-based and host-based schemes to achieve prioritized messages, and evaluate their overhead and effectiveness in the presence of congestion. The network-based scheme relies on prioritization support by network devices, while the host-based scheme implements prioritization in software using a dedicated server. Both schemes can be set up within a site, without assumptions on the external network that connects sites.[4]

For each scheme, we answer four questions: What is required to use it? How to set it up? What is the overhead? How effective is it?

Network-based solution

What is required? The scheme requires devices and appropriate protocols that support prioritization of messages. Prioritized messages are available at several levels:

Layer	Device	Mechanism
IP	IP router	RFC 2474 (DiffServ)
MPLS VPN	Edge router	RFC 2474 (DiffServ)
Ethernet	Switch	IEEE 802.1p

One way to connect sites is via a private leased line, using a modem and an IP router or switch at each end of the line. An alternative cost-effective scheme is to use VPNs, such as MPLS VPNs.

With private leased lines, prioritization is possible via IP or Ethernet solutions. IP solutions were first available using the Type of Service (ToS) bits in the IP header, which were later superseded by the 6-bit DSCP field in IP DiffServ. The DSCP field defines a traffic class for each IP packet, which can be used to prioritize traffic at each network hop. Ethernet solutions are based on the IEEE 802.1p standard, which uses a 3-bit PCP field on an Ethernet frame. Such solutions are available even on commodity low-end switches, where a use case is to prioritize video or real-time gaming traffic in a home network connected to a broadband modem.

With MPLS VPNs, prioritization is available via IP prioritization at the edge routers [1].

How to set it up? The simpler and lower-end devices are configured via web-based user interfaces. Higher-end routers and switches have configuration interfaces based on command lines. We set up traffic prioritization in the Cisco Catalyst 3550 router by configuring it to classify packets based on DSCP bits at ingress and place them accordingly into egress priority queues [8, Chapter 28].

[4] Another scheme is TCP Nice [53], which de-prioritizes traffic. We can conceptually prioritize messages by de-prioritizing all others using TCP Nice, but doing so requires de-prioritizing traffic outside our system. This could be hard and it fails if other systems use UDP.

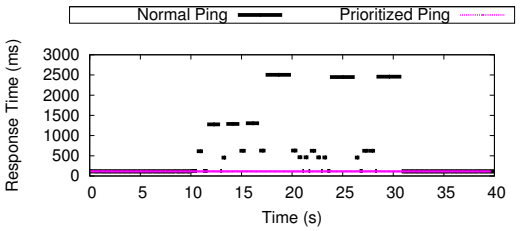


Figure 4: Round-trip delay for regular and prioritized messages using a router with IP DiffServ support.

What is the overhead? We evaluate the overhead of the scheme, by configuring the Cisco Catalyst 3550 router in the local cluster. We measured the round-trip latency of null requests sent from the cluster to EC2 (Ireland), with DSCP bit prioritization enabled and disabled. We found no detectable differences, indicating that the overhead of the prioritization mechanism in the IP router is small.

How effective is it? We perform a simple experiment: two clients in the private cluster periodically measure the round-trip time of their established TCP connection to a machine outside the cluster. One client was configured to set the DSCP field to a priority value, while the other used the default DSCP field. 10 seconds into the experiment, two machines in the cluster generate congestion by running iperf, which sends UDP traffic at a rate of 60 Mbps each to two machines. The congestion continues for 20 seconds and then stops. Figure 4 shows the round-trip latency observed by both clients. We can see that prioritized messages are effective: their latency is unaffected by the congestion. In contrast, congestion causes regular messages to be dropped, which triggers TCP retransmissions—sometimes multiple times—causing large delays due to the TCP back-off mechanism.

Host-based solution

What is required? This scheme requires one or more proxy machines to handle traffic between sites. Messages targeted to machines outside the local site are first sent to a local proxy machine. The proxy forwards the message to a proxy machine in the remote site, which finally forwards the message to the destination. In each proxy, packets are prioritized by placing them into queues according to their DSCP bits. The proxy is a dedicated Linux instance, running SOCKS Dante server processes [4] and using outbound priority queues configured with the HTB packet scheduler in the *tc* tool [7].

How to set it up? To reduce internal traffic at a site, the proxy is placed close to the site's external network connection. Machines are configured to use the proxy, using the SOCKS protocol.

What is the overhead? We measure the round-trip latency of null requests sent between the Virginia and Ireland EC2 locations, with and without the proxy, to determine the extra latency added by the proxy. There was no

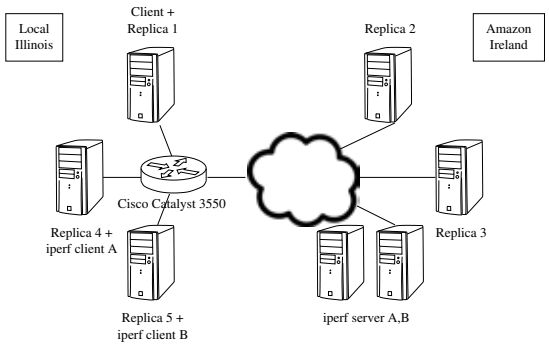


Figure 5: Network setup.

congestion in the network. Without the proxy, the average round-trip latency is 93 ms, while with the proxy, it is 99 ms. The overhead of the proxy is dwarfed by the much larger network latencies across sites.

How effective is it? To evaluate the scheme, we place four machines at each of two EC2 locations (Virginia, Ireland). One machine runs the proxy, one runs two clients, and two run iperf to generate congestion. The experiment is the same as for the network-based solution, except that it uses an extra machine for the proxy, and it uses machines in EC2 only. The results are also similar and demonstrate that the prioritized messages are quite effective (not shown).

Applicability of each solution

The host and network-based solutions are applicable to a small or medium private data center connected by leased lines or MPLS VPNs. Larger data centers or the cloud have larger external bandwidths, and so the host-based scheme creates a bandwidth bottleneck at the proxy, making the network-based scheme a better alternative.

6.3 Storage algorithms

In the next experiments, we consider the performance of the new storage algorithms in Vivace. To do so, we configure Vivace to use either prior algorithms (ABD and Algorithm 2) or the new algorithms (Algorithm 1 and 3). The goal is to understand what are the overheads and benefits of the new algorithms.

The experimental setup consists of five server processes placed in two sites—the local cluster and EC2 Ireland—as shown in Figure 5. Replicas 1, 2, 3 are used as the replica set, with replica 1 placed locally and replicas 2 and 3 placed in Ireland. The local replicas consist of replicas 1, 4, 5, which are in the local cluster. The client is co-located with replica 1. We use the network-based prioritization scheme available in the IP router.

In each experiment, the client issues a series of operations of a given type on a 1 KB object for 20 seconds, and we measure the latency of those operations. For the RW algorithms (ABD and Algorithm 1), the operation types are *read* or *write*; for the RMW algorithms (Algorithms 2

Algorithm	Operation	Min latency	Max latency
ABD Algorithm (prior work)	read	221	228
	write	111	120
Algorithm 1 (new)	read	221	231
	write	113	121
Algorithm 2 (prior work)	execute	221	231
Algorithm 3 (new)	execute	222	231

Figure 6: Round-trip latency with no congestion (in ms).

and 3), the operation type is *execute*.

6.3.1 Overhead under no congestion

The new algorithms are designed by deconstructing some existing algorithms to prioritize critical fields in their messages. This deconstruction increases the number of communication phases of the algorithms, and raises the question of whether they would perform worse than prior algorithms. To evaluate this point, we run an experiment where there is no congestion in the network and we compare performance of the different algorithms.

The results are shown in Figure 6. We find that the algorithms perform similarly, which indicates that the overhead of the extra phases in the new algorithms are not significant. This result confirms the analysis in Section 4.1 when $\delta \ll d$.

6.3.2 Benefit under congestion

In the next experiment, we evaluate the performance of the algorithms under network congestion to understand the benefits of prioritizing critical messages in the new algorithms.

The results are shown in Figure 7. Note that the x-axis has a logarithmic scale. As can be seen, the new algorithms perform significantly better than the equivalent prior algorithms. The continuous congestion causes large latencies in the execution of the prior algorithms. The difference in median latency is over 300ms for all operations. Perhaps more significantly, the differences in the higher percentiles are large. The difference at the 90th percentile for read and write operations is nearly 1s, while for the execute operation it is over 1s. This is particularly relevant because online services tend to have stringent latency requirements at high percentiles: for instance, in Amazon's platform, the latency requirements are measured at the 99.9th percentile [24]. With the use of priority messages, the new algorithms are better suited for satisfying such requirements.

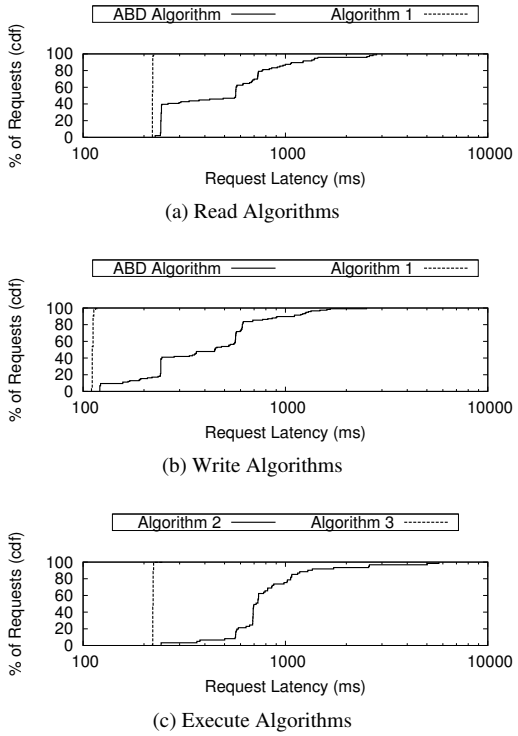


Figure 7: Latency of Vivace under congestion.

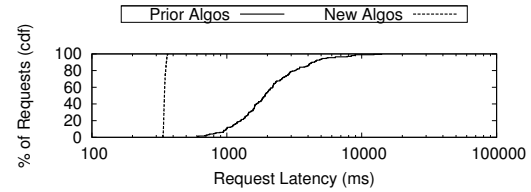
6.4 Effect on a web application

We now consider how the new algorithms in Vivace can benefit a real web application. We use Vivace as the storage system for a Twitter-clone called Twissandra [2], which is originally designed to use the Cassandra [3] storage system. We replace Cassandra with Vivace, to obtain a system that employs the new algorithms with prioritization. More precisely, Twissandra uses Cassandra to store account settings, tweets, follower lists, and timelines. In the modified Twissandra, we store account settings and tweets in Vivace RW objects, and we store follower lists and timelines in Vivace RMW objects.

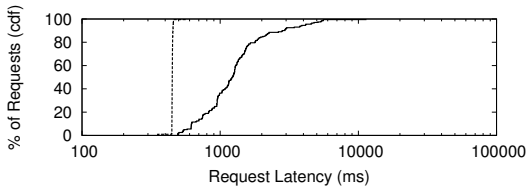
We evaluate the benefit of Vivace's algorithms, by measuring the latency of common user operations in Twissandra. As in Section 6.3.2, we configure Vivace to use the prior and new algorithms, and compare the difference in performance.

We run the experiments with two sites—the local cluster and EC2 Ireland—using the IP router to provide network-based prioritization. A load generator issues a sequence of 200 requests to Twissandra of a given type, one request at a time. We consider three request types: (R1) post a new tweet, (R2) read the timeline of a single user, and (R3) read the timeline of a user's friends. Each of these application requests result in multiple Vivace requests. We measure the latency it takes to process each request while the network is congested with background traffic generated by two machines running iperf (as in other experiments).

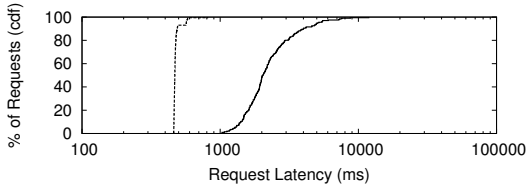
The results are shown in Figure 8. As can be seen, when Vivace is configured to use the new algorithms, the system is much more resilient to congestion. With the prior algorithms, latency for user operations often grew well above 1 second, with a median latency of 2.0s, 1.2s, 2.0s, and maximum latency of 14.1s, 11.3s, 11.9s, for



(a) Post Tweet (R1)



(b) Read User Timeline (R2)



(c) Read Friends Timeline (R3)

Figure 8: Latency of Twissandra-Vivace under congestion.

requests of types R1, R2, R3, respectively. In contrast, with the new algorithms, the median latency was 0.3s, 0.4s, 0.5s, and maximum latency was 0.4s, 0.6s, 0.8s, for the same request types, respectively—showing a significant improvement of using the new algorithms of Vivace under congestion.

7 Related Work

There has been a lot of work on distributed storage systems in the context of local-area networks (e.g., [34, 28, 48, 55, 26, 14, 20, 6, 31, 12, 39, 52, 41]). In contrast, we are interested in a geo-distributed data center setting, which comprises of multiple local-area networks connected by long-distance links. There is also work on distributed file systems in the wide area [54, 44, 49], which is a setting that in some ways resembles geo-distributed systems. These distributed file systems provide a weak form of consistency that requires conflict resolution [44] or other techniques to handle concurrent writes at different locations [54, 49]. In contrast, we provide strong consistency (linearizability), and our focus is on performing well despite congestion of remote links. Dynamo [24] is a key-value storage system where replicas can be located at multiple data centers, but it provides only relaxed consistency and applications need to deal with conflicts. Cassandra [3] is a storage system that combines the design of Dynamo [24] with the data model of BigTable [20]; like Dynamo, Cassandra provides relaxed consistency. PNUTS [23] is a storage system for geo-distributed data centers, but it too resorts to the technique of providing relaxed consistency to address performance problems. COPS [40] is a key-value storage system for geo-distributed data centers, which provides a consistency condition that is stronger than eventual consistency but weaker than strong consistency, because it allows stale reads.

There has been theoretical work on algorithms for read-write atomic objects or arbitrary objects (e.g., [29, 13, 30, 47, 43]). There has also been practical work on Byzantine fault tolerance (e.g., [19, 9, 35, 22, 32]) which considers the problem of implementing a service or state machine that can tolerate Byzantine failures. These algorithms were not designed with the geo-distributed data center setting in mind. In this setting they would see long latencies under congestion, because processes send large messages across long-distance links in the critical path.

In the context of wide-area networks, there have been proposals for more efficient protocols for implementing state machines. The Steward system [15] builds Byzantine fault tolerant state machines for a system with multiple local-area networks connected by wide-area links. They use a hierarchical approach to reduce the message complexity across the wide-area. Mencius [42] is another system that builds a state machine over the wide-area. The use of a multi-leader protocol and skipping allows the system to balance message load according to network conditions. Hybrid Paxos [25] is another way to reduce message complexity. It relies on the knowledge of non-conflicting commands as specified by the application [46, 10, 38]: for instance, commands known to be commutative need not be ordered across replicas, allowing for faster processing. Steward, Mencius, and Hybrid Paxos can reduce or amortize the bandwidth consumed by messages across remote links. Yet, the systems can experience high latency if congestion on these links reduces the available bandwidth to below the message load. Our work addresses this problem by deconstructing algorithms and prioritizing a small amount of critical information needed in the critical path. As long as there is enough bandwidth available for the small fraction of load produced by prioritized messages, bursts of congestion will not slow down our algorithms.

PRACTI [18] separates data from control information to provide partial replication with flexible consistency and data propagation. Vivace also separates certain critical fields in messages, but this is done differently from PRACTI for many reasons. First, Vivace has a different purpose, namely, to improve performance under congestion. Second, Vivace uses different algorithms, namely, algorithms based on majority quorum systems, while PRACTI is based on the log exchange protocol of Bayou [51]. Third, Vivace provides a different storage

service to clients, namely, a state machine [50], while PRACTI provides a more limited read-write service.

Megastore is a storage system that replicates data synchronously across multiple sites. Unlike Vivace, Megastore supports some types of transactions, but it has no mechanisms to cope with cross-site congestion.

Other related work includes peer-to-peer storage systems, which provide weak consistency guarantees rather than linearizability.

8 Conclusion

In this paper, we presented Vivace, a distributed key-value storage system that replicates data synchronously across many sites, while being able to cope with congestion of the links connecting those sites. Vivace relies on two novel algorithms that can overcome congestion by prioritizing a small amount of critical information. We believe that the volume of data across data centers will increase in the future, as more web applications become globalized, which will worsen the problem of congestion across sites. But even if that does not happen, Vivace will still be useful, by allowing remote links to be provisioned less aggressively. More broadly, we believe that geo-distributed systems that make judicious use of prioritized messages will become more relevant, not just for storage systems as we considered here, but also in a wider context.

References

[1] `http://www.cisco.com/en/US/prod/collateral/iosswrel/ps6537/ps6557/prod_white_paper0900aecd803e55d7.pdf` as of Oct 2011.

[2] `https://github.com/twissandra/twissandra`, as of Oct 2011.

[3] `http://cassandra.apache.org`, as of Oct 2011.

[4] Dante – proxy communication solution. `http://www.inet.no/dante/`.

[5] Global MPLS VPN pricing guide. `http://shop2.sprint.com/assets/pdfs/en/solutions/worldwide/taiwan_global_mpls_vpn.pdf` as of Oct 2011.

[6] The Hadoop distributed file system: Architecture and design. `http://hadoop.apache.org/core/docs/current/hdfs_design.html`.

[7] HTB Linux queuing discipline manual—user guide. `http://luxik.cdi.cz/~devik/qos/htb/manual/userg.htm`.

[8] *Catalyst 3550 Multilayer Switch Software Configuration Guide, Cisco IOS Release 12.1(13)EA1*. Mar. 2003.

[9] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *SOSP*, Oct. 2005.

[10] M. K. Aguilera, C. Delporte-gallet, H. Fauconnier, and S. Toueg. Thrifty generic broadcast. In *DISC*, Oct. 2000.

[11] M. K. Aguilera, S. Frolund, V. Hadzilacos, S. L. Horn, and S. Toueg. Abortable and query-abortable objects and their efficient implementation. In *PODC*, 2007.

[12] M. K. Aguilera, W. Golab, and M. A. Shah. A practical scalable distributed B-tree. *VLDB*, 1(1), Aug. 2008.

[13] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. In *PODC*, Aug. 2009.

[14] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *TOCS*, 27(3), Nov. 2009.

[15] Y. Amir, C. Danilov, J. Kirsch, J. Lane, D. Dolev, C. Nita-Rotaru, J. Olsen, and D. Zage. Scaling Byzantine fault-tolerant replication to wide area networks. In *DSN*, 2006.

[16] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *JACM*, 42(1), Jan. 1995.

[17] J. Baker et al. Megastore: Providing scalable, highly available storage for interactive services. In *Conference on Innovative Data Systems Research*, Jan. 2011.

[18] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *NSDI*, 2006. Extended version available at `http://www.cs.utexas.edu/users/dahlin/papers/PRACTI-2005-10-extended.pdf`.

[19] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *OSDI*, Feb. 1999.

[20] F. Chang et al. Bigtable: A distributed storage system for structured data. In *OSDI*, Nov. 2006.

[21] G. Chockler and D. Malkhi. Active disk paxos with infinitely many processes. *Distributed Computing*, 18(1), July 2005.

[22] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *NSDI*, Apr. 2009.

[23] B. F. Cooper et al. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB*, Aug. 2008.

[24] G. DeCandia et al. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP*, 2007.

[25] D. Dobre, M. Majuntke, M. Serafini, and N. Suri. HP: Hybrid paxos for WANs. In *European Dependable Computing Conference*, Apr. 2010.

[26] J. R. Douceur and J. Howell. Distributed directory service in the Farsite file system. In *OSDI*, Nov. 2006.

[27] E. Gafni and L. Lamport. Disk paxos. *Distributed Computing*, 16(1), Feb. 2003.

[28] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, Oct. 2003.

[29] S. Gilbert, N. Lynch, and A. Shvartsman. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In *DSN*, June 2003.

[30] S. Gilbert, N. A. Lynch, and A. A. Shvartsman. RAMBO: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4), Dec. 2010.

[31] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable distributed data structures for internet service construction. In *OSDI*, 2000.

[32] J. Hendricks. Efficient Byzantine fault tolerance for scalable storage and services. Technical Report CMU-CS-09-146, Carnegie Mellon University, School of Computer Science, July 2009.

[33] M. P. Herlihy and J. M. Wing. Acm toplas. *ACM Trans. Program. Lang. Syst.*, 12, July 1990.

[34] J. H. Howard et al. Scale and performance in a distributed file system. *TOCS*, 6(1), Feb. 1988.

[35] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative Byzantine fault tolerance. In *SOSP*, Oct. 2007.
[36] T. Kraska, G. Pang, M. J. Franklin, and S. Madden. MDCC: Multi-Data Center Consistency. 1203.6049, Mar. 2012.
[37] L. Lamport. The part-time parliament. *TOCS*, 16(2), May 1998.
[38] L. Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, Mar. 2005.
[39] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *ASPLOS*, Oct. 1996.
[40] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen. Don't settle for eventual: Stronger consistency for wide-area storage with COPS. In *SOSP*, Oct. 2011.
[41] P. Mahajan et al. Depot: Cloud storage with minimal trust. In *OSDI*, 2010.
[42] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for wans. In *OSDI*, 2008.
[43] J.-P. Martin and L. Alvisi. A framework for dynamic Byzantine storage. In *DSN*, June 2004.
[44] L. B. Mummert, M. R. Eblig, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *SOSP*, Dec. 1995.
[45] J. Nielsen. *Designing Web Usability: The Practice of Simplicity*. New Riders Publishing, 1999.
[46] F. Pedone and A. Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15(2), Apr. 2002.
[47] R. Rodrigues and B. Liskov. Rosebud: A scalable Byzantine-fault-tolerant storage architecture. Technical Report TR/932, MIT LCS, Dec. 2003.
[48] Y. Saito, S. Frolund, A. Veitch, A. Merchant, and S. Spence. FAB: building distributed enterprise disk arrays from commodity components. In *ASPLOS*, Oct. 2004.
[49] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mhalingam. Taming aggressive replication in the Pangaea wide-area file system. In *OSDI*, Dec. 2002.
[50] F. B. Schneider. Implementing fault-tolerant services using the state machine approach : A tutorial. *ACM Computing Surveys*, 22(4), Dec. 1990.
[51] D. B. Terry et al. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, Dec. 1995.
[52] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *SOSP*, Oct. 1997.
[53] A. Venkataramani, R. Kokku, and M. Dahlin. TCP nice: A mechanism for background transfers. In *OSDI*, Dec. 2002.
[54] R. Y. Wang and T. E. Anderson. xFS: A wide area mass storage file system. In *Workshop on Workstation Operating Systems*, Oct. 1993.
[55] S. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI*, Nov. 2006.

Practical Hardening of Crash-Tolerant Systems

Miguel Correia
IST-UTL / INESC-ID
Lisbon, Portugal

Daniel Gómez Ferro
Yahoo! Research
Barcelona, Spain

Flavio P. Junqueira
Yahoo! Research
Barcelona, Spain

Marco Serafini
Yahoo! Research
Barcelona, Spain

miguel.p.correia@ist.utl.pt, {danielgf, fpj, serafini}@yahoo-inc.com

Abstract

Recent failures of production systems have highlighted the importance of tolerating faults beyond crashes. The industry has so far addressed this problem by hardening crash-tolerant systems with *ad hoc* error detection checks, potentially overlooking critical fault scenarios. We propose a generic and principled hardening technique for Arbitrary State Corruption (ASC) faults, which specifically model the effects of realistic data corruptions on distributed processes. Hardening does not require the use of trusted components or the replication of the process over multiple physical servers. We implemented a wrapper library to transparently harden distributed processes. To exercise our library and evaluate our technique, we obtained ASC-tolerant versions of Paxos, of a subset of the ZooKeeper API, and of an eventually consistent storage by implementing crash-tolerant protocols and automatically hardening them using our library. Our evaluation shows that the throughput of our ASC-hardened state machine replication outperforms its Byzantine-tolerant counterpart by up to 70%.

1 Introduction

Distributed systems in production require dependability mechanisms to avoid extended periods of unavailability, violations of data integrity, and other undesirable consequences of faults. Coordination systems, such as ZooKeeper [24], and storage systems, such as GFS [18] and Bigtable [11], all include mechanisms to prevent faults from disrupting their operation. These systems are all designed to tolerate crashes, since crashes are observable and are common in production environments. Crash-tolerance assumes that processes fail in a silent manner and never send incorrect messages. This assumption, however, may be violated in presence of undetected corruptions of the internal state of a faulty process. The impact or even the sheer occurrence of such data corruptions can be very hard to ascertain because they are not properly detected by the system. One infamous failure occurred in the Amazon S3 storage service in July 2008 [2]. It required taking the whole system down for an incremental restart and resulted in an 8-hour outage. Post-mortem failure diagnosis concluded that:

> *A handful of messages had a single bit corrupted such that the message was still intelligible, but the system state information was incorrect. We used MD5 checksums throughout the system (but not) for this particular internal state information. (...) When the corruption occurred, we did not detect it and it spread throughout the system causing the symptoms described above [2].*

Other failures have been traced back, or hint to, corruptions of the internal state of a process; for example, replicas diverging in the Chubby lock service [10], data loss in Ma.gnolia [31], and data corruption in S3 [3]. Publicly known failures are often related to services exposed to external users. The scale of the problem possibly goes far beyond these known cases, since companies often keep data corruption incidents confidential.

Practical distributed systems often use CRCs, MD5 hashes, or other error detection codes to detect data corruptions. Due to the lack of viable principled approaches, however, developers often find it difficult to reason about appropriate placements of checks into their code. *Ad hoc* error detection might not cover important fault scenarios, making the system susceptible to severe outages.

In this paper, we propose a principled hardening approach that specifically targets *realistic* faults in distributed systems. We surveyed faults that have been observed in post-mortem analysis of production datacenter systems or through fault injection campaigns. We then focused on the ones that can be tolerated through distributed fault tolerance techniques, which comprise many of the faults we surveyed. We observed that these faults manifest at the process level as state corruptions

or control-flow corruptions, rather than as "adversarial" process behavior. Our fault injection experiments on Paxos confirm this observation.

Based on the above observations, we propose a new fault model for Arbitrary State Corruption (ASC) faults, modeling the effect of realistic faults at the level of a process of a distributed system. ASC admits that faults change the state of a faulty process to an arbitrary value and modify the execution flow of the process code. ASC faults can occur an unbounded number of times.

We then propose a *hardening* technique that guarantees error isolation: an ASC-faulty process does not propagate erroneous state to other processes through erroneous messages. Either the faulty process itself detects the error and crashes, or the recipient detects a faulty message and drops it. The designer of a distributed protocol can thus focus on the tolerance of crashes and message omissions, which are simpler to handle, and is relieved from the burden of applying error detection checks and reasoning about guarantees when introducing them. The hardened version of a process is still a single process, which does not need to be replicated over multiple physical machines. Hardening is achieved by adding redundancy to the state and the computation of the original process, as well as to the messages it sends. No trusted component is assumed: the hardening state and computation can also be corrupted.

We automated our hardening technique by designing a library, called PASC, that wraps processes and transparently hardens them. Our hardening algorithm only assumes that processes communicate with each other through message-passing. Using PASC, we obtained ASC-tolerant versions of protocols with different consistency, fault tolerance, and scalability properties, simply by hardening the processes of corresponding crash-tolerant protocols, with only minor modifications. We implemented the Paxos protocol [28], a subset of the ZooKeeper API on top of Paxos, and a scalable eventually consistent storage, with acceptable performance overhead (around 17% less throughput for ZooKeeper).

A safer but more expensive alternative to our approach is tolerating Byzantine faults, where faulty processes can turn into adversaries and make any theoretically possible action to harm other correct processes. Byzantine-fault tolerance (BFT) protocols implement strongly consistent state machine replication (SMR). Beyond data corruptions, BFT tolerates software bugs, intrusions and other malicious process behavior under the assumption that a quorum of correct replicas is always available. This can be achieved by using diverse replicas [17], but development costs and the difficulty of achieving independence of failures in practice [26] prevent the use of design diversity in many cases. Despite the good performance of existing BFT algorithms [9, 27, 39, 43], and the existence of prototypes of complex systems using BFT [1], the industry has not adopted BFT as a viable solution to dependable systems, to the best of our knowledge.

ASC-hardening differs from BFT in a number of key aspects. First, it considers security orthogonal to fault tolerance, which is what most practical systems do, and focuses on the latter. Second, it does not need replication to prevent error propagation. Existing approaches preventing error propagation when processes fail in a Byzantine manner use variants of SMR, resulting in higher costs [23]; furthermore, many distributed systems do not need replication of all their processes, or use replication with weaker consistency than strongly consistent SMR. The third key difference is that any process of a distributed system can be ASC-hardened, including clients of client-server and multi-tier architectures; this is an important property in practical distributed systems [22], but achieving it with a Byzantine fault model can be very complex [33]. Finally, ASC-hardening is significantly more efficient than BFT. Our micro-benchmarks show that ASC-tolerant Paxos delivers up to 70% higher throughput than the BFT protocol of Castro and Liskov [9].

In this paper we make five main contributions:

- A new process-level fault model, ASC, representing realistic data corruptions;
- A sound technique to harden crash-tolerant processes against ASC faults;
- A library, PASC, that enables developers to automatically harden crash-tolerant protocols;
- An evaluation of ASC-hardened versions of Paxos, of a subset of the ZooKeeper API, and of an eventually consistent store;
- A validation of the ASC model and of the coverage of PASC by injecting code and state corruptions in our Paxos implementation.

2 Realistic faults

Understanding failure behavior of computer systems has been the topic of decades of research, which traditionally focused on monolithic mainframe systems (see for example the work by Jim Gray on the Tandem system [19]). We surveyed recent failures of distributed production systems, which are often built using commodity and low-end servers. We also included results of large-scale studies on faults in such systems. We did not consider security issues because, in most production system, they are not handled using fault tolerance techniques.

We selected representative reports that are publicly available; it is not our intention to provide an exhaustive or comprehensive survey of all failure events in production systems. This discussion, however, is sufficient

N.	Failure description	Class of faults	Crash	ASC	Byz.
1	Bugs in several database management systems with different designs shown to corrupt data in study [17]	Software development faults	no		yes †
2	Google App Engine unavailable due to a bug in the GFS Master: a malformed file handle caused successive crashes and recoveries (July 9 2009) *(http://groups.google.com/group/google-appengine/msg/ba95ded980c8c179)*	Software development faults	no		maybe †
3	Paxos replica state corruption due to an illegal memory access made by errant code included in the codebase [10]	Software development faults	no		maybe †
4	Two Yahoo! client applications had bugs caused by developers that misinterpreted the semantics of the ZooKeeper API [40]	Software development faults	no		
5	Administrators configured Nagios to monitor ZooKeeper by pinging certain TCP ports; each ping created a thread that was not closed, leading to too many threads *(https://issues.apache.org/jira/browse/ZOOKEEPER-880)*	Operation faults	no		
6	DNS misconfiguration caused machine names to clash, preventing a ZooKeeper instance running at Yahoo! from continuing to work (July 2009) [40]	Operation faults	no		
7	Incorrect NIC configurations caused frequent leader elections and system instability in a ZooKeeper instance running at Yahoo! [40]	Operation faults	no		
8	Google App Engine datastore unavailability due to instability in the cluster, which overloaded the Bigtable repository keeping the locations of chunks; timeouts started to expire and all requests started to fail (May 25 2010) *(http://groups.google.com/group/google-appengine-downtime-notify/msg/e9414ee6493da6fb)*	Operation faults	no		
9	Facebook offline for 2.5 hours due to automatic error correction mechanism that overloaded a database with queries (September 23 2010) *(http://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919)*	Operation or Software development faults	no		
10	Paxos replica inconsistency despite consistent execution logs, probably caused by a hardware memory error corrupting the replica state [10]	Hardware fault	no		yes
11	Uncorrectable corruption of data in ECC DRAMs shown to happen in large-scale study [38]	Hardware faults	no		yes
12	Amazon S3: A new but defective load balancer corrupts some relayed messages until it is taken offline, after 36 hours (June 20-22, 2008) *(https://forums.aws.amazon.com/thread.jspa?threadID=22709)*	Hardware faults	no		yes
13	Amazon S3 unavailability due to corruption of a single bit in a few messages, which propagated errors to multiple servers; this generated a high volume of gossiping of the system state, bringing request processing close to a halt (July 20, 2008) *(http://status.aws.amazon.com/s3-20080720.html)*	Hardware faults	no		yes
14	Sun servers fail randomly due data corruption in memory caches caused by cosmic rays [44]	Hardware faults	no		yes
15	Ma.gnolia looses half terabyte of its customers' data due to file system corruption (Jan. 09) *(http://getsatisfaction.com/magnolia/topics/ ma_gnolia_data_recovery_status)*	Hardware faults	no		yes
16	Sidekick unavailability, apparently due to disk failure; the data took several days to be reconstructed (October 2009) *(http://latimesblogs.latimes.com/technology/2009/10/microsoft-says-lost-sidekick-data-will-be-restored-to-users.html)*	Hardware faults	no		yes
17	Undetected data corruptions in disks, often due to buggy firmware, shown to happen in large-scale study [5]	Hardware faults	no		yes
18	SSD devices found to lose data due to incorrect firmware (2009) *(http://www.dailytech.com/Update+Intel+Confirms+SSD+Data+Corruption+Issue+Suspends+Shipments+Pending+Firmware+Update/article15827.htm)*	Hardware faults	no		yes
19	IOMMU chipset of some AMD boards undetectably corrupts data due to a bug, which is only activated in some specific hardware/software configurations *(https://bugzilla.kernel.org/show_ bug.cgi?id=7768)*	Hardware faults	no		yes
20	Google services partially unavailable for approximately 1 hour due to fire in a datacenter (2010) *(http://mobilelocalsocial.com/2010/google-data-center-fire-returns-world-wide-404-errors/)*	Hardware faults	yes ⋆		
21	Los Angeles International Airport law enforcement database unavailable due to a faulty NIC that caused a packet storm and took the entire LAN offline (August 2007) *(http://www.crn.com/news/networking/201801022/nic-card-soup-gives-lax-a-tummy-ache.htm)*	Hardware faults	yes ⋆		

Table 1: Recent failure events and large-scale dependability studies. In several cases there was a cascade of faults, so the class refers to the root cause. The three columns on the right indicate the ability of distributed crash, ASC and Byzantine fault-tolerant systems to tolerate the faults that caused the events († these faults are tolerated only if replicas fail independently, e.g. through design diversity; ⋆ only using multiple networks/locations).

to illustrate that real failures can be quite complex, with process faults that are more severe than just crashes.

Failure reports. Table 1 loosely organizes failure events and studies in classes. In some cases the assignment to a class is speculative due to lack of detailed information.

Software development faults, or bugs, are a pervasive class of faults (rows 1-4 in the table). Even in well-tested software, bugs whose activation depends on intricate combinations of states and inputs and/or heavy loads are usually present [4] and can lead to data corruption in databases (rows 1, 3) or cause effects like cyclic crashes and recoveries (row 2). Bugs cannot be tolerated using commonly available distributed fault-tolerant techniques. In principle, some development faults (row 1) may be tolerated using BFT if processes are replicated and each replica contains different software, so the bug does not affect all of them [17]. However the use of diversified design is often not feasible.

Some of the faults of Table 1 cannot be tolerated even with BFT. For example, operation faults are human errors

made during system configuration and at runtime. Row 5 shows a case in which a monitoring tool was configured in a way that was not predicted by the software developers, leading to the consumption of threads, eventually causing abnormal operation. Several other examples are shown in rows 6-9. These faults are hard to tolerate automatically regardless of the fault model adopted because they can easily result in correlated failures ("complex human-machine interactions (...) remain a challenge" [4]). Client-side software development faults (row 4) or bugs in the implementation of trusted fault tolerance libraries (row 9) are also not tolerated by BFT.

Despite these inherent limitations, there is a large fraction of faults that can be handled using distributed fault tolerance. Hardware faults are known to cause corruption of data in RAM (rows 10-13), memory caches (row 14), and hard disks (rows 15-17). Some hardware faults are actually caused by development faults in hardware components or, more often, in hardware-related software such as firmware or drivers. It is well known that firmware can cause data corruptions in hard disks; such corruptions can only be detected using additional checks at the operating system or application level (row 17). Other classes of block storage devices, like solid-state drives (SSD), may also corrupt data (row 18). Most current servers use Single Error Correction, Double Error Detection (SECDED) ECC DRAMs, but a recent large-scale study has found that their error rates are orders of magnitude higher than previously reported and that several uncorrectable errors are detected (row 16). It is not possible to know how often undetected memory errors violate data integrity without being reported; some analytical model indicates that such errors may be not uncommon [14]. The study reports a correlation between CPU load and rate of detected memory errors, indicating that data corruptions may also occur in the data path to and from memory. For example, motherboard development faults have been found to cause undetected data corruptions (row 19).

Data-center wide failures due to environmental reasons such as fires (row 20) have also caused massive outages. Network failures are known to corrupt the content of messages, but sometimes may make large networks unavailable (row 21).

After reviewing these failure reports, we conducted a closer analysis of realistic faults whose effects can be tolerated using BFT without design diversity, as for example hardware errors. Our goal was understanding how these faults can be observed by processes of a distributed system. There are three main observations we could draw from this analysis.

Observation 1: State corruptions cause most failures. The tolerable failures of Table 1 are caused by corruptions of the *state* of some process rather than arbitrary, or even adversarial, process behaviors. This is consistent with recent studies that have investigated in detail the effect of systematic injection of a very large number of *code* corruptions in a popular Linux distribution [20] and a group membership service [7]. Both studies report that randomly corrupted instructions may corrupt part of the process state or divert the control-flow of the process. Gu *et al.* [20] report that "*detailed tracing of crash dumps indicates that random error injections can corrupt several instructions in a sequence*". This is because the processor (an Intel CPU) may interpret the instruction stream as a sequence of random instructions if it starts reading it from an incorrect location. The same can occur in case of a faulty jump to a location that is not the beginning of a valid instruction. However, these faults are not dangerous, as discussed by the authors: "*As a result, the system executes an invalid sequence of instructions, which is very likely to cause quick (i.e., short latency) crash*". The paper also includes examples of the injected code corruptions, in which a crash always occurs immediately after the corrupted instructions are executed due to invalid arguments or references. The cases of error propagation across processes discussed by Basile *et al.* [7] are also caused by direct state corruptions or by single corrupted instructions that corrupt the state.

Observation 2: Control-flow faults are dangerous. According to the aforementioned fault injection experiments on the Linux code, code corruptions inverting the outcome of branch operations are the most most likely to generate incorrect outputs [20]. The danger of control-flow faults has long been known in highly dependable systems and has lead to the development of a range of techniques for control-flow checking (see for example [32]). Therefore, such faults need to be considered as possible sources of non-crash process faults.

Observation 3: Reported failures are due to transient faults. The relevant failures reported in Table 1 do not seem to be caused by permanent faults being repeatedly activated. Indeed, many faults of Table 1 are explicitly categorized as soft, or transient, errors, something that was already observed by Gray [19]. Hardware-level fault injection campaigns indicate that permanent process faults are likely to manifest as quick crashes [29]. Some memory modules have scrubbing facilities for detecting potential permanent faults proactively [38]. If detected but uncorrectable memory errors in DRAM modules are not discarded by the operating system or application, they might cause non-crash process faults [30]. However, it is common practice to shut down the server and replace the faulty memory module in these cases [38]. Disk subsystems commonly employ specific techniques to recover from some permanent faults such as bad sectors.

3 The Arbitrary State Corruption model

We now introduce our Arbitrary State Corruption (ASC) model, which formalizes realistic faults based on the three observations we made in the previous section. First, the fault model should represent state corruptions, including those caused by error propagation. Second, it should also model control-flow faults leading to incorrect jumps to some correct operations of the same faulty process. Third, it should consider transient errors that could sporadically appear over time, without assuming that the root cause of the problem is diagnosed. The ASC model includes crashes, which represent a trivial case for hardening. We will focus our discussion on faults that cannot be modeled as simple crashes.

The corruption of even a single variable can propagate to multiple other variables, depending on the way the process state is organized. A similar argument arises when considering control flow faults, whose analysis may require deriving complex application-specific control-flow graphs based on the low-level execution blocks of the code [32].

While sophisticated analyses can be conducted if the low-level details of the distributed system implementation are known, we want our hardening technique to be applicable to generic distributed systems with minimal knowledge. We only assume that processes communicate via unreliable message passing and hold a local, initially correct state. Every time a process receives a message, it calls the corresponding event handler, which can modify the local state and produce one or more output messages. We also want the hardening algorithm to be independent of its low-level implementation.

ASC fault model. The ASC fault model simplifies the matter by taking a conservative approach. It considers all data that is locally accessed by a process as its state, and it assumes that a fault may arbitrarily change the values of any number of variables. Faults can also make the control flow jump to any instruction of the process.

Formally, the model abstracts each process π of a distributed system as a set of guarded commands of the form $\langle G(m_{in}), B(S, m_{in})\rangle$, where S is the process state, m_{in} an input message, $G(m_{in})$ is a boolean activation condition and $B(S, m_{in})$ is an event handler. Event handlers modify the state and produce a list of output messages. Processes are deterministic: any input message activates at most one event handler.

Definition 1 (ASC fault) *An* ASC fault *occurring at a process π either causes π to crash or assigns an arbitrary value to any variable of its process state S, possibly modifying the control flow of π and causing it to execute from an arbitrary process instruction.*

ASC faults can occur multiple times and at any time, but not arbitrarily often: at most one fault can occur during the execution of a single event handler. In our use cases, event handling occurs in the order of a few milliseconds at most, a conservative bound given the fault frequencies reported in the literature (e.g. [38]).

Data integrity. The model assumes a data integrity property that corresponds, in the ASC model, to the cryptographic assumptions limiting the strength of an adversary in the Byzantine fault model.

Data integrity is protected by replicating each variable of S with a variable of a *replica state R*. The following (slightly simplified) definition refers to corruptions of variables in S; the same property also holds if variables in R are corrupted.

Definition 2 (Fault diversity) *Immediately after a fault modifies the value of a variable v of a state S, the value of v in S is different from the value of v in the replica state R until either (i) v is modified by an assignment, or (ii) the replica of v is assigned to a new value obtained by reading v.*

Fault diversity makes it possible to detect data corruptions by comparing replica variables; however, it does not mandate that the two variables will remain different forever, making detection challenging. The above definition admits two cases in which a corrupted variable v takes the same value as its replica. For example, assume that the current correct value of v and its replica is 0, but an ASC fault assigns v the erroneous value 5. The two replicas are different immediately after the fault. The event handler in execution may use the incorrect value of v to determine the value of other variables, and then re-initialize v to 0, making the corruption undetectable again. Such execution is admitted by our definition of fault diversity. If the comparison is executed after v is re-initialized, the corruption is not detected and error propagation can occur, so the execution time of this check is key. Hardening must minimize the number of checks while providing error isolation guarantees.

4 ASC hardening

This section presents our ASC hardening technique. An overview of the steps executed by a fully ASC-hardened process is given in Algorithm 1; a more formal and detailed pseudocode can be found in the accompanying technical report [13]. Instead of directly discussing Algorithm 1, we introduce checks incrementally starting from a simple protocol example. We discuss failure scenarios to guide the derivation of our ASC-hardening technique. Due to space limitation, we cannot exhaustively consider all fault scenarios; we refer to our technical report for a complete correctness proof [13].

We target systems where processes communicate by sending message through unreliable channels, which can duplicate messages, drop them, and send them to the wrong recipients. Message corruptions respect fault diversity if message fields are replicated.

ASC hardening guarantees the following property.

Property 1 (Error isolation) *Let m be a message sent by a faulty hardened process, and assume that some of the variables whose value is included in m have an incorrect value when the message is sent. If a correct hardened process receives m, then it discards m without modifying its local state. If a faulty hardened process receives m and modifies its local state according to m, then it crashes before sending any output message.*

The correct value of a variable is defined inductively for every received message m: either m is dropped and its receipt has no local effect, or m is processed by updating all required variables without faults. Error isolation prevents both *direct* error propagation, from faulty processes to correct processes, and *indirect* error propagation, caused by faulty processes that send incorrect messages in response to another incorrect message.

Error isolation guarantees that, after a fault, a process with a corrupted state will only expose a benign behavior. In many cases the process eventually crashes as a consequence of executing some internal check; however, a fault may corrupt values at any time, even in the "last mile" before the message is actually sent. These faults are harmless because *all* incorrect messages a faulty process will send after a state corruption will be discarded by their recipients.

At a high level, hardening guarantees error isolation by ensuring that, when an output message is sent, the value of any corrupted variable in S does not match with the value of its replica in R. A sender forms the content of the messages using variables in S, and its verification code using the corresponding replicas in R. This use of message verification enables the receiver to discard not only messages corrupted by the network, but also messages built using corrupted sender state.

Hardening is not trusted: ASC faults can corrupt any hardening variable introduced in Algorithm 1, as for example the replica state R, and let the control flow start from any instruction of the hardened process code.

For illustration purposes, we consider the example of a client of a distributed data store. Algorithm 2 illustrates one event handler of the client. It reads a value r from the store, modifies a state variable v, and writes back v.

Simple checks. There are two straightforward ways to harden a protocol: *message integrity* and *state integrity* checks. Message integrity checks use message codes (e.g., CRC codes) against network corruptions and the boolean activation condition to verify that the correct event handler is being executed. Every time a message is received containing, for example, a value read from the store, it is verified using the message code and discarded if needed. State integrity checks replicate the process state either using a full copy or using codes. Let us call S the original state and R the replica state. Every time a variable is read from S, these checks compare its value with the one stored in R. If a mismatch occurs, a process can preserve integrity in this case by crashing.

The limit of these checks is that they assume that a newly updated variable can only get corrupted after being replicated. Consider for example a fault corrupting the variable *new* during the execution of the event handler. Even if *new* is replicated, there is no redundant information protecting this newly computed value yet. The incorrect value propagates both internally, by overwriting both S and R, and externally to the data store. The simple checks are not sufficient to prevent this problem.

Hardening against faulty computation. Our ASC-hardening technique addresses the previous problem by executing the event handler twice. The first execution accesses S, the second R. This prevents error propagation of incorrect values between the two states. If the state prior to receiving the READ-REP message of Algorithm 2 is correct, at most one of the two executions may be impacted by a fault, so one of the two states will be correct.

The client builds the WRITE output message using the value of v in S and its message code using the replica of v in R. The receiver of a message, the store process in the example, detects incorrect values contained in the message by verifying its code.

Checking the checkers. Our hardening technique does not assume the existence of trusted components such as trusted voters. Checks can also fail, as we discuss with the following example. We have previously assumed that the state before receiving the READ-REP message is correct. Consider now the alternative case where a previous fault had already corrupted v and its replica, resulting in a latent error. For example, assume that v and its replica had a correct value 0, but previous faults gave them the incorrect values 7 and 8, respectively. When the message is received, the state integrity check typically detects the corruption. However, a fault might invert or skip the outcome of the check, making the value of v appear as correct and letting both executions of the event handler produce the same incorrect output.

Our ASC-hardening addresses this problem by checking state integrity during both executions of the event handler. If at least one execution of the event handler is correct, it either computes correct new values using correct inputs or crashes. Instead of modifying S directly, the first execution of the event handler now uses copy-

Algorithm 1: Overview of a hardened event handler.

1 Message integrity check;
2 First execution of the event handler, using state S – during the execution, check state integrity of variables the first time they are read, store modifications to S in a copy-on-write buffer N, and store the identifiers of the modified variables in U;
3 First at-least-once gate;
4 First at-most-once gate;
5 Second execution of the event handler, using replica state R – during the execution, check state integrity of variables the first time they are read, and store the identifiers of modified variables in U';
6 Second at-most-once gate;
7 Second at-least-once gate;
8 For each variable with identifier in U, apply changes from N to S and check that the identifier is in U', crashing otherwise;
9 Check that $U = U'$ and crash otherwise;
10 First and second at-least-once checks;
11 Message integrity check;

Algorithm 2: Example event handler pre-hardening.

12 **upon** receive ⟨READ-REP, r⟩ from the store process
`// v, new are state variables`
13 **if** $v > 5$ **then** $new \leftarrow r + v + 5$;
14 **else** $new \leftarrow r + v$;
15 $v \leftarrow new$;
16 send ⟨WRITE, v⟩ to the store;

on-write access to S, storing incremental changes in a buffer N. In the example, a copy of v is added to N to store its new value. The second execution of the event handler can thus access the original value of S for its state integrity check. The incremental changes of N are made persistent in S after the second execution has completed. The sets of variable identifiers U and U' are used to prevent incorrect state updates in line 8. Our ASC-hardening also verifies message integrity twice.

Handling control-flow faults. We now discuss how to guarantee correctness in spite of control-flow faults. A fault might result in the incomplete execution of an event handler: for example, after updating the value of the variable *new*, the client might not update v, sending the old value of v to the store. Executing the handler multiple times is also incorrect. For example, the client might add r to v twice and send this incorrect value to the store.

Algorithm 1 handles these faults using what we call control-flow gates and checks. For each control flow gate, hardening introduces a different pair of control-flow variables, c and its replica c', and uses a distinct label L. Initially, c and c' are set to a value different from L. The fact that a control flow variable is set to L marks the fact that the process has reached a given point in the execution flow of the hardened event handler. Control flow variables are replicated to prevent faults from incorrectly setting both of them to L. For simplicity, in the following discussion we use the same names for all control flow variables and labels.

An *at-most-once* gate $\mathscr{G}$ is a sequence of three high-level instructions. First, if $\mathscr{G}.c \neq \mathscr{G}.c'$ or $\mathscr{G}.c = L$ then the process crashes. Else, $\mathscr{G}.c$ is set to L and $\mathscr{G}.c'$ is set to L. Note that the full gate cannot be executed twice, even in presence of a fault. These gates are used to prevent situations where instructions of an event handler are executed twice, for example adding r to v twice. The hardened process uses two separate instances of at-most-once gates, at lines 4 and 6 of Algorithm 1.

An *at-least-once* gate $\mathscr{G}'$ is a sequence of two assignments of $\mathscr{G}'.c$ and $\mathscr{G}'.c'$ to L, respectively. An at-least once check verifies that the corresponding gate has been reached by checking that $\mathscr{G}'.c = \mathscr{G}'.c' = L$ and crashing otherwise. There are two separated instances of at-least-once gates in Algorithm 1, at lines 3 and 7. The checks for both gates are at line 10.

We have discussed that the following condition must hold when output messages are sent after the hardened event handler has completed its execution: if the value of a variable in S is incorrect, then it does not match with its replica in R. For variables in S or R whose current value has been last modified by a fault, the condition holds by definition of fault diversity. Therefore, we focus on showing that the condition holds for variables determined by instructions; in this case, fault diversity might not be sufficient.

Consider the case where some variable of S is modified by an instruction. Variables in S are modified only in line 8. Let t_8 be the last time when an instruction of line 8 is executed. If no fault occurs before t_8, then both executions of the even handler in lines 2 and 5 complete correctly before t_8. Variables could be corrupted by executing instructions of the event handler after t_8 if a fault occurs. However, one of the at-most-one gates would lead to a crash.

If no fault occurs after t_8, the at-least once checks of line 10 either crashes the process or guarantees that lines 3 and 7 are executed before t_8. Let t_3 and t_7 be the last time when the two gates are executed, respectively.

If no fault occurs after t_3 or before t_7, then the second execution of the event handler is executed correctly once. The at-most-once gates ensure that this second execution is not repeated. After the second execution is completed, the updates of all variables of R are correct, and the correct identifiers of these updated variables are in U'. Any variable of S that is supposed to be updated but has an

incorrect value can thus be detected as faulty by comparing it with its correctly updated replica in R. The checks on U and U' of line 9 ensure that only the right variables of S are updated.

The last case is the one where, if no fault occurs before t_3 or after t_7, the first execution of the event handler updates N and U correctly. By definition of t_3, no fault can occur after t_3 and lead to executing instructions of line 2 again. After t_7, the state S is updated in line 8. If some value of N is corrupted by a fault after being correctly updated, this can be detected due to fault diversity. The checks on U and U' of lines 8 and 9 guarantee that the right updates are executed.

5 The PASC library and use cases

The PASC library automates the hardening of distributed protocols. It is a runtime execution environment inside which user-defined protocols are executed. The user is required to provide PASC with a specification of the *protocol state S*, the *event handlers*, and the *messages* used by the protocol. Each of these components corresponds to an interface that must be implemented by user-defined classes. The library is currently implemented in Java.[1]

The implementation of message-passing primitives is external to hardened processes, and thus to PASC, so the user has no restriction of how to implement them. During initialization, the user must create a new PASC runtime object and pass it references to all the user-defined protocol classes. The protocol is actually run by passing every received message to a method of the runtime, which selects the correct event handler and returns output messages. The runtime takes care of transparently running the hardened event handler.

Particular care is needed when defining the protocol state, a class whose fields are the state variables. PASC transparently builds a replica R of the protocol state S. PASC needs to track accesses to variables in order to implement copy-on-write updates, execute state integrity checks, and build the sets U and U'. We use the names *get-v* and *set-v* for the methods accessing the variable v, where v is a unique label identifying the variable. A "variable" can be any subset of the protocol state. During initialization, PASC encapsulates the state class into a dynamically generated class that intercepts all calls to getters and setters. PASC requires that event handlers use only getters and setters of the state class for reading from and writing to state variables. Checking variables before reading them is a major source of performance overhead, which grows with the size of the variables. The user can reduce this overhead by defining many fine-grained getters instead of few coarse-grained ones. For example, a getter returning an array is more expensive to call than one returning only a single element of the array.

Messages are transparently replicated and verified, but the user needs to specify how the message replica is stored in the message, and how to verify messages. By default, replica messages are CRC32 codes. We used TCP for message passing.

Use case: Paxos state machine replication. We used PASC to implement an ASC-tolerant version of the Paxos SMR protocol, a common fault-tolerant protocol implemented by many practical system (e.g. [10]).

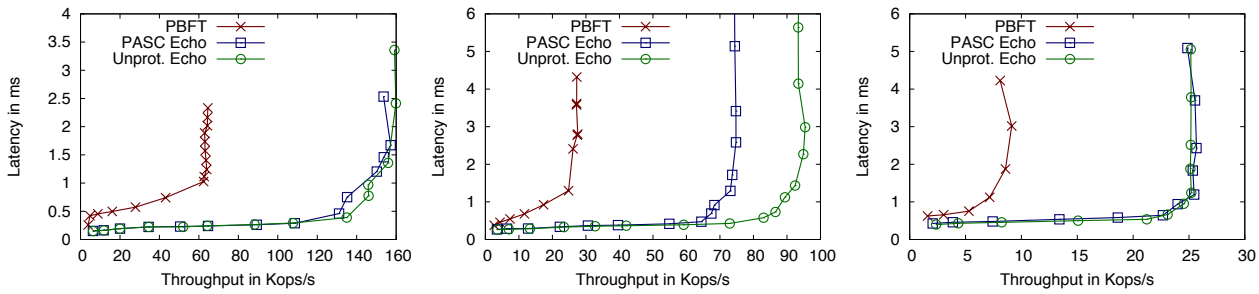
The Paxos protocol uses a leader to propose an execution order of requests. It tolerates the presence of multiple concurrent leaders, but only ensures progress in normal periods with only one leader. The critical path of requests in normal periods is as follows. The leader collects and orders requests from the clients, sending a sequence of operations to all other processes. If a process accepts the order proposed by the leader, it sends an acknowledgement to all other processes. A process executes requests in the sequence indicated by the leader only after receiving acknowledgements from a majority of processes. Clients deliver the first reply they receive.

Paxos tolerates crashes but not data corruptions. The corruption of a single variable of a single process can lead to the unrecoverable loss of the history of executed operations. For example, if the current ballot number a replica stores gets corrupted, then the recovery of a new leader may complete in a state that is inconsistent with the previously committed state. The new leader can then incorrectly overwrite committed state.

In the ASC-tolerant version of Paxos, both client and replica processes are hardened with PASC. The same number of replicas as for crash-tolerance is sufficient: at least $2f+1$ to tolerate f faulty replicas. Like in Zookeeper, the replication protocol itself detects if the current leader is faulty and uses a leader selection algorithm only to elect a new leader.

PASC Paxos guarantees error isolation for the messages exchanged by the consensus layer. This ensures that correct processes execute operations in a consistent order, and that faulty replicas do not process incorrect messages before sending apparently correct messages. Achieving full ASC tolerance, however, requires also handling corruptions to the state of the state machine, not only of consensus. Handling faulty state machines boils down to two simple changes of PASC Paxos compared to the original Paxos. Clients cannot deliver the first reply they receive from any replica; instead, they must wait for a quorum of $f+1$ consistent replies to make sure that at least one reply comes from a correct process. Given the presence of $2f+1$ replicas, a quorum of correct replicas is available. Furthermore, processes periodically take checkpoints of the state machine to enable garbage col-

[1] The library is open source, see https://github.com/yahoo/pasc



(a) Request and reply size: 0kB (b) Request and reply size: 1kB (c) Request and reply size: 4kB

Figure 1: Micro-benchmarks of hardening: Latency-throughput curves.

lection of the operation log. These checkpoints can be undetectably corrupted, so they need to be validated by other processes. This is done by simply exchanging a digest of the state and waiting for $f+1$ confirmations of the digest. There are known techniques to efficiently compute incremental digests of states, as for example the Merkle trees proposed in [9]. With these changes, the replicated state machine does not need to run inside the PASC runtime.

Use case: Eventually consistent storage. ASC hardening can be used as a building block to implement ASC-tolerant state machine replication, but it is not restricted to it. Our second use case shows that it is possible to harden fault-tolerant algorithms with different consistency guarantees.

We designed a simple key-value store, called SimpleKV, which replicates data with relaxed consistency to scale throughput and latency. It implements a single-writer multiple-readers register, a fundamental abstraction in distributed computing. Clients have write access to their own key and read access to all keys, which are replicated by multiple distributed data stores. Each replica holds a copy of all keys. Clients write requests to $f+1$ stores for persistence before returning, where f is the number of faulty stores that must be tolerated. SimpleKV does not need to write to a majority quorum of stores; this ensures scalability of write operations. Clients add a client-local timestamp to write requests. A store overwrites the local value of a key only if it receives a write with a higher timestamp than any other write previously observed for that key. Read operations are served by a single store, which ensures scalability.

SimpleKV guarantees eventual consistency [45]: in periods when no client invokes write operations, all data stores eventually hold the same latest value for all keys. Whenever a data store applies a write to its local state, it forwards the new value in the background to the other data stores, following the process ID order.

6 Evaluation

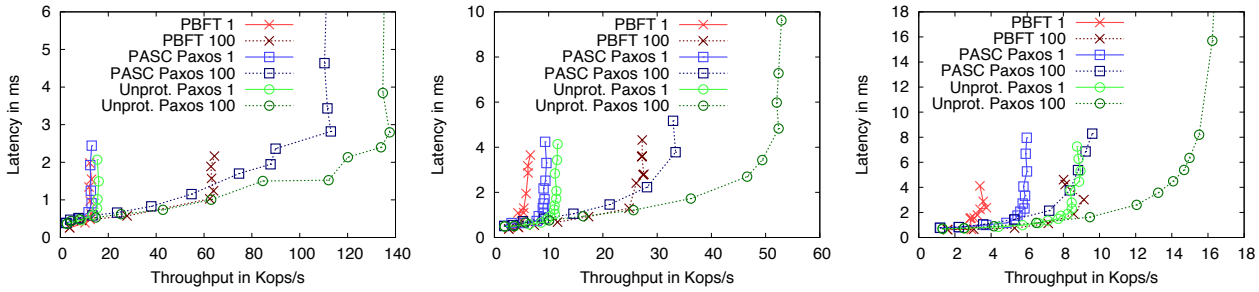
6.1 Performance of PASC

The ASC-tolerant use case protocols show the flexibility of our hardening approach. However, ASC-hardening must also be feasible from a performance viewpoint. In this section, we show that this is the case. We compare the performance of our use cases with and without ASC hardening. PASC has an option to turn off hardening, running just the original processes.

Our evaluation setup consists of servers equipped with a quad-core 2.5 GHz CPU, 16 GB of RAM, and a Linux with kernel 2.6.18. We use a Gigabit network. All protocols are configured to tolerate one fault, so $f=1$. Protocol clients are run on dedicated machines, different from the ones used for replicas and data stores. Each client sends a new request as soon as its previous request is completed. All measurements are started after the clients have issued a few thousands of requests and the system has reached steady-state performance. We disable multicast for all protocols because the network does not support it. All plots show averages over five runs; we observed negligible variance. We control throughput by increasing or reducing the number of clients.

Hardening micro-benchmarks. ASC hardening guarantees error isolation without using state machine replication (SMR). This simplifies the hardening of many practical distributed systems, where processes are loosely coupled. In our eventually consistent SimpleKV store, for example, ASC hardening is sufficient to prevent the propagation of incorrect data and messages; SMR is not needed for fault tolerance.

Hardening with Byzantine faults is much more expensive. Existing work hardens scalable crash-tolerant systems against f Byzantine faults using variants of SMR and replicating each process (*host*) on $3f+1$ different servers (*guards*) [23]. This leads to higher replication costs and increases complexity.



(a) Request and reply size: 0kB (b) Request and reply size: 1kB (c) Request and reply size: 4kB

Figure 2: Micro-benchmarks of state machine replication: Latency-throughput curves.

Figure 1 compares the performance of hardening a single process. We use micro-bechmarks, where the hardened process receives a request and sends a reply, without making any computation. We consider request and reply sizes of 0, 1 and 4 kilobytes. For the Byzantine model, we provide the performance of Castro's implementation of the PBFT protocol [9] as a reference value since no implementation of the SMR protocol of [23] is publicly available. Networking is the main bottleneck in the 0 kB and 4 kB cases due to high number of messages and message size, respectively. The overhead of PASC is negligible in these cases. The cost of message verification is more evident in the 1 kB case. In all cases, executing a full BFT SMR protocol is substantially more expensive: PASC improves throughput by about 2.5x.

SMR micro-benchmarks. For state machine replication, we compare our ASC-hardened implementation of the Paxos protocol and Castro's implementation of the PBFT protocol, which can be seen as the enhancement of Paxos for the Byzantine fault model: it also uses a leader to order requests and only executes requests when they have been agreed upon (at least tentatively). Paxos requires $2f+1$ replicas to tolerate f faults, regardless of hardening; PBFT needs $3f+1$ replicas, although it can be extended to use $3f+1$ replicas for agreement and $2f+1$ for execution [47].

We stress the system by considering all-writes workloads where the system must agree on each request. All protocol implementations use batching with congestion window, a technique introduced by PBFT. The leader batches incoming requests as long as agreement on a previous batch of requests is ongoing. We extended this mechanism by letting the leader start agreement on a batch anyway if the number of requests in the current batch reaches a certain threshold. We show results for maximum batch sizes 1 (no batching) and 100 since larger batch sizes did not significantly improve performance. We execute the same micro-benchmarks as the previous experiments. For reference, a typical average request size for ZooKeeper is around 1 kB [25].

The latency-throughput curves of the different protocols are reported in Figure 2. Using batching results in a net throughput improvement and it does not impact latency significantly. Adding CRCs to messages is common in crash-tolerant protocols, so we evaluated its impact on the throughput of Paxos and found that it is negligible. PASC Paxos outperforms PBFT in maximum throughput between 70% and 10%, depending on the size of the message. Paxos outperforms PASC Paxos significantly only with larger messages: message verification becomes more costly, storing messages requires transferring more data due to our use of copy-on-write, and larger areas of memory need to be checked.

The minimal latency when a single client submits requests sequentially is reported in Figure 3. The latency of the three protocols is similar and very low. This is because PBFT uses tentative executions to send replies with the same analytical latency as Paxos [9]. Unlike PBFT, our Paxos implementation processes messages using a pipeline of different stages, which results in slightly higher latency.

Figure 4 reports the steady-state JVM user memory usage. The occupation grows with the loads of the system so we consider runs with peak throughput. Both in case of Paxos and PASC Paxos, the occupation grows rapidly between 0 kB and 1 kB, and remains almost flat for 4 kB. While the replication of the protocol state does result in higher memory occupation, the absolute overhead is small. The relative difference between the two protocols remains around 43-58% regardless of the message size. The state machine is not encapsulated inside PASC and its state is not replicated, so the memory overhead would not grow if we would consider running a state machine on top of PASC Paxos.

ZooKeeper. Micro-benchmarks are good for stress-testing state machine replication protocols. However, these protocols are never run stand-alone. For a more realistic assessment of the performance cost of these sys-

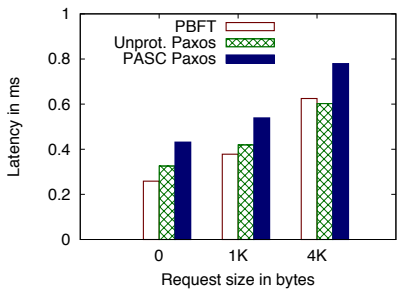


Figure 3: Single request latency

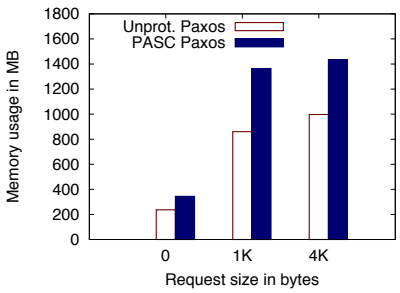


Figure 4: Memory occupation

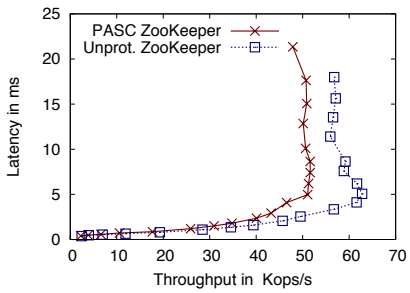


Figure 5: ZooKeeper benchmark

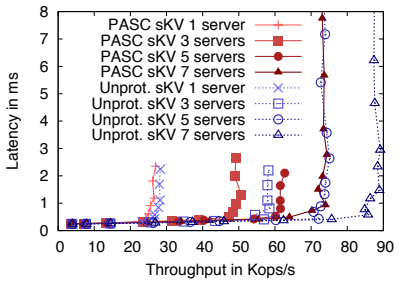


Figure 6: SimpleKV performance and scalability.

tems, we implemented a subset of the ZooKeeper API on top of Paxos. ZooKeeper exposes a tree-like data structure that is stored in memory. We chose the subset of commands that can be implemented using vanilla state machine replication: *create*, *delete*, *exists*, *get data*, *set data*, *get children*. Handling ephemeral nodes and sessions would have required handling timeouts, which are non-deterministic. Watchers require clients to handle replies while they may have other pending operations; this does not directly fit into the concept of well-formed run. For our implementation we use the actual data structure used internally by ZooKeeper to store its state, which is called DataTree, and build glue code to make it communicate with Paxos.

In order to test the system we created a custom ZooKeeper benchmark. Clients issue requests choosing a random command. Each command operates on a random node of a data tree of depth up to 5. Each internal node of the tree has up to 5 children. Figure 5 compares the performance of our ZooKeeper implementation running on Paxos and PASC Paxos. Both in terms of latency and throughput, the overhead of using PASC is less significant than in micro-benchmarks. The peak difference in maximum throughput between the two versions is around 17%. Note that the ZooKeeper state is not replicated in PASC Paxos, since ZooKeeper is a state machine.

SimpleKV. We run SimpleKV with an increasing number of data stores to study the scalability of the protocol with and without PASC. We use a 20/80 benchmark, where clients continuously issue operations that are write in the 20% of the cases and reads in the remaining 80%. The size of requests and replies is 1 kB. Figure 6 shows the latency-throughput curve resulting from our experiments. Both the original SimpleKV protocol and PASC SimpleKV scale almost linearly with the number of servers. A BFT-hardened version of SimpleKV would have used most of the processes for replication, using SMR groups, rather than for better performance.

6.2 Coverage of the ASC model

We now study how random faults affect the operation of our Paxos implementation; the results support our claim that the ASC model is a proper approach to capture the key properties of these faults. We follow closely the established approach of works like Gu *et al.* [20] and Basile *et al.* [7]: injecting single bit flips. We increase the likelihood of activating the injected faults by using stack traces or targeting *core classes*, which are selected by profiling the execution of Paxos. The core classes are the top ten CPU- and memory-intensive classes (considering all their instances) in each of the following categories: Paxos classes, PASC-runtime classes, Java library classes, messaging classes (we used Netty), and JVM-internal classes.

We inject faults at the Paxos leader because it is the process with the highest likelihood of propagating errors. After injecting faults, we wait approximately one minute for its manifestation before interrupting the experiment. We report results of our injections during the normal execution of the protocol, where a single leader is present in the system. Some fault injections result in a leader crash and trigger a subsequent recovery. We also reproduced worst-case scenarios where the leader crashes and some other replica has a corrupted state, and found no case of error propagation using PASC.

We consider four types of injections, corrupting *bytecode*, *binary code*, *pointers* and *primitive values*.

Bytecode injection: In each run, we select a random

	Code corruptions				State corruptions			
	bytecode		binary code		pointer		value	
	unprot.	hardened	unprot.	hardened	unprot.	hardened	unprot.	hardened
Undetected error propagation	0	0	3	0	66	0	27	0
Detection through hardening	-	0	-	1	-	164	-	166
Crash or hang	956	1028	684	635	2165	2024	136	42
Total number of runs	1818	1818	1047	1047	4000	4000	1237	1237

Table 2: Results of fault injection experiments

method among core classes and flip one random bit on its bytecode before this is loaded by the JVM. We disable JVM bytecode verification at loading time.

Binary injection: At a random time during the execution of the protocol, we take the stack trace. We then inject bit flips around 1 to 10 random return addresses selected among those found in the stack, at a random location within a range of 400 bytes.

Pointer and value injection: At a random time during the execution of the protocol, we corrupt one random field of a random instance of a random core class. We restrict to non-primitive values for pointer injections and to primitive values for value injections.

Results. Table 2 summarizes the results of our fault injection experiments. We partition runs into three classes, based on the effect of the injected fault: runs where *undetected error propagation* occurs; runs with no error propagation where an internal error of the faulty process or an incorrect message is *detected through hardening*; runs where no error is propagated and no internal error or incorrect message is detected through hardening, but the faulty process *crashes or hangs* anyway; and runs where the injected fault has no effect (not shown).

Our main conclusion is that hardening is both necessary and sufficient to prevent undetected error propagations in our experiments: error propagation does occur, but only without hardening. Most code injections result in crashes if the error is activated. A few cases of potential error propagation due to code injection occur if processes are not protected, but ASC-hardening is sufficient to guarantee error isolation. State corruptions in pointers and values are much more likely to generate error propagation. These findings support our choice of modeling all random data corruptions as ASC faults.

7 Related work

The rigorous design and proof of correctness of fault-tolerant distributed algorithms require the formalization of a system model. In the selection of such a model there is always a tension between several facets: how well it describes reality, the classes of problems it allows solving, and the efficiency of the algorithms that assume it [37]. Our ASC fault model represents a new tradeoff for practical fault-tolerant systems. It was inspired by Dijkstra's seminal work on self-stabilization, where the whole system state can transition to arbitrary values [15]; however, the goal of ASC hardening is isolating faulty processes rather than converging to a correct global state.

Some theoretical work proposed transformations to improve the fault tolerance of distributed algorithms from crashes to Byzantine faults, mainly targeting crash-tolerant agreement algorithms where processes communicate by broadcasting information over multiple communication rounds [12], or synchronous systems [35]. NewTOP makes strong synchrony assumptions and may not tolerate the failures of two replicas [34]. Baldoni *et al.* use failure detection modules tailored to a round-based consensus protocol in order to detect some faults beyond crashes [6]. ASC hardening only makes very basic assumptions about the structure of distributed processes; it is not restricted to agreement protocols, it does not depend on synchrony assumptions, and it does not require designing protocol-specific checks. Nysiad is a more generic hardening technique that does not require synchrony for progress; as discussed in our evaluation, it hardens against Byzantine faults, but it uses a variant of state machine replication as building block [23]. Byzantine fault detection is a viable alternative to detect integrity violations after they have occurred, but it does not guarantee error isolation [21].

Hypervisor-based approaches run multiple replicas of a process inside the same physical machine. Traditionally, these approaches targeted crash or fail-stop models that do not encompass ASC faults [8]. Furthermore, unlike ASC hardening, they assume failure-independence of local process replicas and a trusted hypervisor.

Techniques for error detection come at least from the 50s with Hamming's work. Taylor et al. proposed adding redundant data to data structures to detect corruptions of their instances [41]. Detection of control-flow corruptions in software has also a long tradition. Tront et al. proposed a simple technique to detect control-flow corruptions that force software to jump to a different subroutine [42]. The technique uses a tag or signature to check in which routine processing should be at the mo-

ment. Several variations of this idea have later appeared.

ASC hardening uses knowledge about the structure of distributed programs to guarantee end-to-end error isolation properties and execute very few checks. There exist many other hardening techniques in the literature that do not assume knowledge of the hardened program. For generality, they trade higher performance overhead, lack of end-to-end guarantees, or a less comprehensive fault model. For example, SWIFT is a compiler-based technique that targets a single event upset, where a single bit is flipped once [36]. Unlike our hardening technique, SWIFT assumes that all memory errors are detected by hardware error correcting codes. SWIFT executes twice every binary instruction computing new values, and compares the two obtained values immediately afterwards; this potentially results in high overhead, although a performance comparison with unprotected code is not given in [36]. ASC hardening executes full event handlers twice too, but it does not execute comparisons after every operation. Unlike ASC, SWIFT does not protect against faults corrupting values after the comparison and before storing them in memory, so it does not fully guarantee error isolation. Finally, SWIFT's control flow signatures detect a subset of the control flow errors covered by our technique.

An older low-level hardening technique is AN-coding, where instructions generate encoded variables from encoded variables [16]. AN-codes protect the algorithm against some subsets of hardware faults (e.g. operator corruptions, but not exchanged operators). Software Encoded Processes (SEP), proposed in [46], run on top of an encoded interpreter, which uses AN-codes and other coding techniques. The fault coverage of SEP is probabilistic and depends of the number and distribution of the bit flips. Being low-level, SEP leads to high overhead, with slowdowns between 2.2x and 25x.

Yoo *et al.* use local checkpoints to harden protocols against crashes [48]. ASC-hardening is orthogonal: it contains error propagation with ASC faults but requires explicit handling of crashes and message omissions.

8 Conclusion

We proposed ASC-hardening as a novel, sound approach to systematically protect the integrity of distributed systems against realistic faults. Hardening prevents error propagation across processes by converting state corruptions into process crashes and message omissions. We have implemented a library, PASC, to transparently harden processes and evaluated it using a few realistic examples: state-machine replication, a key-value store implementation, and a subset of the ZooKeeper coordination service. Our performance evaluation showed that PASC enables excellent performance while inducing an acceptable penalty; for our ZooKeeper benchmark, we obtained over 50k ops/s and roughly a 17% throughput drop compared to the unprotected version. Our fault injection experiments also showed that PASC is able to effectively prevent error propagation upon data corruptions. Preventing error propagation is critical to avoid massive outages in many production systems.

Acknowledgements

We would like to thank our shepherd Andreas Haeberlen, Jon Howell, and the anonymous reviewers for the valuable feedback. This work has been partially supported by the SRT-15 project (ICT-257843), funded by the European Community. Marco Serafini and Flavio P. Junqueira also acknowledge support from the INNCORPORA - Torres Quevedo Program from the Spanish Ministry of Science and Innovation, co-funded by the European Social Fund. Miguel Correia was partially supported by FCT through the Multi-annual PIDDAC Program funds, project RC-Clouds and scholarship SFRH/BSAB/992/2010.

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 *Proc. of USENIX OSDI* (2002), pp. 1–14.

[2] AMAZON. Amazon S3 availability event: July 20, 2008. `http://status.aws.amazon.com/s3-20080720.html`, July 2008.

[3] AMAZON. S3 data corruption? `https://forums.aws.amazon.com/thread.jspa?threadID=22709`, June 2008.

[4] AVIZIENIS, A., LAPRIE, J.-C., RANDELL, B., AND LANDWEHR, C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing 1*, 1 (Jan-Mar 2004), 11–33.

[5] BAIRAVASUNDARAM, L. N., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., GOODSON, G. R., AND SCHROEDER, B. An analysis of data corruption in the storage stack. *ACM Transactions on Storage 4*, 3 (2008), 1–28.

[6] BALDONI, R., HELARY, J.-M., AND RAYNAL, M. From crash fault-tolerance to arbitrary-fault tolerance: Towards a modular approach. In *Proc. of IEEE DSN* (2000), pp. 273–282.

[7] BASILE, C., LONG, W., KALBARCZYK, Z., AND IYER, R. Group communication protocols under errors. In *Proc. of IEEE SRDS* (2003), pp. 35–44.

[8] BRESSOUD, T. C., AND SCHNEIDER, F. B. Hypervisor-based fault tolerance. *ACM Transactions Computer Systems 14*, 1 (February 1996), 80–107.

[9] CASTRO, M., AND LISKOV, B. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems 20*, 4 (2002).

[10] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live: An engineering perspective. In *Proc. of ACM PODC* (2007), pp. 398–407.

[11] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *Proc. of USENIX OSDI* (2006), pp. 205–218.

[12] COAN, B. A. A compiler that increases the fault tolerance of asynchronous protocols. *IEEE Transactions on Computers 37*, 12 (Dec. 1988), 1541–1553.

[13] CORREIA, M., FERRO, D. G., JUNQUEIRA, F., AND SERAFINI, M. Models and algorithms for ASC hardening and a correctness proof. Yl-2011-003, Yahoo! Labs, 2011.

[14] DELL, T. J. A white paper on the benefits of Chipkill - correct ECC for PC server main memory. Tech. rep., IBM Microelectronics Division, 1997.

[15] DIJKSTRA, E. W. Self-stabilizing systems in spite of distributed control. *Communications of ACM 17* (November 1974), 643–644.

[16] FIORIN, P. Vital coded microprocessor principles and application for various transit systems. In *IFAC/IFIP/IFORS Symposium* (1989), pp. 79–84.

[17] GASHI, I., POPOV, P. T., AND STRIGINI, L. Fault tolerance via diversity for off-the-shelf products: A study with SQL database servers. *IEEE Transactions on Dependable and Secure Computing 4*, 4 (2007), 280–294.

[18] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *Proc. of ACM SOSP* (2003), pp. 29–43.

[19] GRAY, J. Why do computers stop and what can be done about it? In *Proc. of IEEE SRDS* (1986), pp. 3–12.

[20] GU, W., KALBARCZYK, Z., IYER, R. K., AND YANG, Z. Characterization of Linux kernel behavior under errors. In *Proc. of IEEE DSN-DCCS* (2003), pp. 459–468.

[21] HAEBERLEN, A., KOUZNETSOV, P., AND DRUSCHEL, P. Peerreview: Practical accountability for distributed systems. In *Proc. of ACM SOSP* (2007), pp. 175–188.

[22] HAMILTON, J. Observations on errors, corrections, and trust of dependent systems. `http://perspectives.mvdirona.com/2012/02/26/ObservationsOnErrorsCorrectionsTrustOfDependentSystems.aspx`, Mar. 2012.

[23] HO, C., VAN RENESSE, R., BICKFORD, M., AND DOLEV, D. Nysiad: Practical protocol transformation to tolerate Byzantine failures. In *Proc. of USENIX NSDI* (2007), pp. 175–188.

[24] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proc. of USENIX ATC* (2010).

[25] JUNQUEIRA, F., REED, B., AND SERAFINI, M. Zab: High-performance broadcast for primary-backup systems. In *Proc. of IEEE DSN* (2011), pp. 245–256.

[26] KNIGHT, J. C., AND LEVESON, N. G. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering 12*, 1 (1986), 96–109.

[27] KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., AND WONG, E. Zyzzyva: Speculative Byzantine fault tolerance. In *Proc. of ACM SOSP* (2007), pp. 45–58.

[28] LAMPORT, L. The part-time parliament. *ACM Transactions Computer Systems 16*, 2 (May 1998), 133–169.

[29] LI, M.-L., RAMACHANDRAN, P., SAHOO, S. K., ADVE, S. V., ADVE, V. S., AND ZHOU, Y. Understanding the propagation of hard errors to software and implications for resilient system design. In *Proc. of ACM ASPLOS* (2008), pp. 265–276.

[30] LI, X., HUANG, M. C., SHEN, K., AND CHU, L. A realistic evaluation of memory hardware errors and software system susceptibility. In *Proc. of USENIX ATC* (2010), pp. 6–16.

[31] MA.GNOLIA. Ma.gnolia data recovery status. `http://getsatisfaction.com/magnolia/topics/ma_gnolia_data_recovery_status`, February 2009.

[32] MAHMOOD, A., AND MCCLUSKEY, E. J. Concurrent error detection using watchdog processors-a survey. *IEEE Transaction on Computers 37*, 2 (February 1988), 160–174.

[33] MERIDETH, M., IYENGAR, A., MIKALSEN, T., TAI, S., ROUVELLOU, I., AND NARASIMHAN, P. Thema: Byzantine-fault-tolerant middleware for web-service applications. In *Proc. of SRDS* (2005), pp. 131–140.

[34] MPOELENG, D., EZHILCHELVAN, P., AND SPEIRS, N. From crash tolerance to authenticated Byzantine tolerance: A structured approach, the cost and benefits. *Proc. of IEEE DSN* (2003), 227–236.

[35] NEIGER, G., AND TOUEG, S. Automatically increasing the fault-tolerance of distributed systems. In *Proc. of ACM PODC* (1988), pp. 248–262.

[36] REIS, G., CHANG, J., VACHHARAJANI, N., RANGAN, R., AND AUGUST, D. SWIFT: Software implemented fault tolerance. In *Proc. of IEEE/ACM CGO* (2005), pp. 243–254.

[37] SCHNEIDER, F. B. What good are models and what models are good? *Distributed systems (2nd Ed.)* (1993), 17–26.

[38] SCHROEDER, B., PINHEIRO, E., AND WEBER, W.-D. DRAM errors in the wild: A large-scale field study. In *Proc. of ACM SIGMETRICS* (2009), pp. 193–204.

[39] SERAFINI, M., BOKOR, P., DOBRE, D., MAJUNTKE, M., AND SURI, N. Scrooge: Reducing the costs of fast Byzantine replication in presence of unresponsive replicas. In *Proc. of IEEE DSN* (2010), pp. 353–362.

[40] SONG, Y. J., JUNQUEIRA, F., AND REED, B. BFT for the skeptics. In *Proc. of BFTW*^{3} (2009).

[41] TAYLOR, D. J., MORGAN, D. E., AND BLACK, J. P. Redundancy in data structures: Improving software fault tolerance. *IEEE Transactions on Software Engineering 6*, 6 (1980), 585–594.

[42] TRONT, J. G., ARMSTRONG, J. R., AND OAK, J. V. Software techniques for detecting single-event upsets in satellite computers. *IEEE Transactions on Nuclear Science 32*, 6 (Dec. 1985), 4225–4228.

[43] VERONESE, G. S., CORREIA, M., BESSANI, A. N., C., L., AND VERISSIMO, P. Efficient Byzantine fault tolerance. *IEEE Transactions on Computers*. To appear.

[44] VIJAYKRISHNAN, N. Soft errors: Is the concern for soft-errors overblown? In *Proc. of IEEE ITC* (2005), pp. 2–12.

[45] VOGELS, W. Eventually consistent. *Communications of the ACM 52*, 1 (2009), 40–44.

[46] WAPPLER, U., AND FETZER, C. Software encoded processing: Building dependable systems with commodity hardware. In *Proc. of SAFECOMP* (2007), pp. 356–369.

[47] YIN, J., MARTIN, J.-P., VENKATARAMANI, A., ALVISI, L., AND DAHLIN, M. Separating agreement from execution for Byzantine fault tolerant services. In *Proc. of ACM SOSP* (2003), pp. 253–267.

[48] YOO, S., KILLIAN, C., KELLY, T., CHO, H. K., AND PLITE, S. Composable reliability for asynchronous systems: Treating failures as slow processes. In *Proc. of USENIX ATC* (2012).